
# A Common Architecture for Simulation Viewing over Multiple Protocol Environments

THESIS

Glenn G. Jacquot
Captain, USAF

AFIT/GCS/ENG/96D-11



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-11

# A Common Architecture for Simulation Viewing over Multiple Protocol Environments

THESIS

Glenn G. Jacquot
Captain, USAF

AFIT/GCS/ENG/96D-11

AFIT/GCS/ENG/96D-11

# A Common Architecture for Simulation Viewing over Multiple Protocol Environments

THESIS

Presented to the Faculty of the
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Glenn G. Jacquot, B.S.
Captain, USAF

December 17, 1996

*Acknowledgements*

I would like to thank Dr. Hartrum. His guidance and direction proved to be an invaluable benefit, especially during the tougher times of the effort. I would also like to thank the members of my committee, Dr. Lamont and Lt Col Stytz, for their contributions towards my research.

Many thanks to all of my classmates, especially to both Capt David Wells, for helping me try to comprehend the SBB, and Capt Ken Stauffer, who was always willing to accept the abuse that I dished out during my most frustrated moments.

Finally, I would like to thank my parents for always being there to support me, no matter how difficult I got.

Glenn G. Jacquot

*Table of Contents*

*List of Figures*

*Abstract*

This study discusses an architecture used for displaying data results in a graphical format. This data is obtained via internet connection or from a direct connection to a battlefield simulator known as BattleSim.

This architecture utilizes several graphics packages for displaying the output along with two different approaches to receiving commands from the user. Furthermore, the design was built to easily incorporate other graphics packages and communication protocols in order to increase its portability.

# A Common Architecture for Simulation Viewing over Multiple Protocol Environments

## *I. Introduction*

### *1.1 Background*

In an effort to improve combat performance and to test out various strategies, the Air Force has often relied on large-scale simulations. These simulations are usually in the form of war games and full-scale field training exercises. Due to the amount of manpower and material required to carry out a successful exercise, however, these efforts have proven to be very costly to the military.

As a result, the military saw computers as the economical solution to this problem. Furthermore, as computer technology improves at an exponential rate, the military's ability to have bigger and more realistic simulations increases. However, as the size of these simulations increased, the greater the need for speed became. Due to the limited ability of single processor machines, multiprocessor platforms quickly became the focus. Efforts were made to utilize this hardware with the software used on them becoming the key to success. One of these software systems created for these multiprocessor systems is BattleSim (3).

Designed at the Air Force Institute of Technology (AFIT), BattleSim was developed to investigate parallel processing to speed up simulation. It uses discrete event simulators which are designed so that the objects are responsible for updating their own states. At the start of every turn, the event queue determines if any events need to be handled during that turn. If so, those

events are sent to the objects responsible for handling them. Upon receiving the event, the object executes the necessary functions, which may include scheduling new events for accomplishing the assigned task. This cycle then repeats itself until the simulation ends.

With a basic file editor, the user generates a battle scenario and then executes it on BattleSim. The results generated by BattleSim are reasonably accurate, but the output is in a text format only. Most users want to "see" their results so a need for a visual interface existed. As a result, the Visual Interactive Simulation Interface Tool (VISIT) was created.

Designed by DeRouchey (3) and modified by Looney (7), VISIT serves as a tool for graphically depicting the results of BattleSim. A slight modification to BattleSim provided three modes of operation. The first mode allows BattleSim to pipe its output directly to VISIT while the second allows VISIT to receive the data from a file that holds the output from BattleSim. Since the output in the file is not in the same format required for VISIT to use, it is necessary to use a file converter called SIMTOGE (SIMulation To Graphics Engine) which converts the output file into a format recognized by VISIT. This mode makes it possible to test either VISIT or BattleSim without requiring the other to be in operation. The third option results in no battle output and is often used for performance analysis.

Looney's accomplishment was to add the ability for a user to interact with BattleSim and VISIT. His approach to proving the success of his effort was to add the ability to stop, rewind, then playback a simulation at the user's discretion. This was accomplished by having BattleSim save its state at regular time intervals, which are defined by the user. Upon the user's command, BattleSim will back up to the state whose time-stamp is just prior to the user's time request. From

there, the user can restart the battle. This allows the user to review any portion of the battle as often as desired.

Numerous simulation systems were being developed nationwide so a need existed for a network to be designed so that these simulations could interact with each other. By doing so, it would no be longer necessary for these systems to be transported to a single location in order for them to run together. After several attempts, the Distributed Interactive Simulation (DIS) was created (8). Instead of creating a network, several organizations, including Advanced Research Projects Agency (ARPA), the Joint Warfighting Center (JWFC), the Defense Modeling and Simulation Office (DMSO), the Simulation Training and Instrumentation Command (STRICOM) of the Army, and the Federal Aviation Administration (FAA), decided to design a network protocol. This protocol would ensure that every player in the simulation was speaking the same language while, at the same time, providing the flexibility of incorporating a wide variety of systems. This protocol was based upon packets, referred to as Protocol Data Units (PDUs). These PDUs contain generic information about a given player, such as location and icon type, but it is up to each individual system on how that information will be used.

Since DIS was not available when BattleSim and VISIT were developed, both systems were based on an Oakridge architecture. Although similar to DIS in that it uses packets for communication, it is not as popular as DIS. As a result, there are fewer systems with which BattleSim and VISIT can interact. To overcome this limitation, the next step was to convert BattleSim from an Oakridge protocol to a DIS protocol. This conversion increased BattleSim's ability to interact with other simulation systems. After BattleSim was converted, it was necessary to convert VISIT

so it could handle a DIS protocol. The final conversion required was that of SIMTOGE so that it produces DIS PDUs, not the Oakridge Packets.

Another graphical interface designed at AFIT is the BattleBridge, which is used in the Graphics Laboratory (11). This interface uses more realistic graphics that are based on the Performer Graphics package. As a result, this system has more realistic looking simulations, but, since Performer is not on every computer, this package also limits BattleBridge's portability. In addition to the conversion of VISIT to DIS, it would also be in AFIT's best interest to integrate VISIT with BattleBridge. The two interfaces differ in that VISIT has a rewind/playback capability that BattleBridge lacks, but BattleBridge has already incorporated the DIS protocol that VISIT needs. This integration eliminates the need to support two separate interfaces that serve the same purpose. Furthermore, AFIT will have a single graphical interface that contains every capability developed at AFIT.

### *1.2 Problem*

The Oakridge Protocol, on which VISIT and BattleSim are based, is not a commonly used protocol in AFIT. Most network-related applications, including BattleBridge, are based on the Distributed Interactive Simulation (DIS) protocol. As a result, the portability of VISIT and BattleSim are severely limited. This handicap limits the ability of the Air Force to use this application world-wide, if necessary. It is, in effect, limited to use only at AFIT and only with those applications based on the Oakridge Architecture. However, with the development of DIS, it is now feasible to have a system capable of interacting with other simulation systems, both local and world-wide. Therefore, the conversion of VISIT and BattleSim to a DIS protocol is necessary for both systems to have the flexibility required for interacting with sources outside of AFIT.

As stated earlier, the Parallel Laboratory and the Graphics Laboratory both support their own graphical interfaces. BattleBridge is simply a graphical viewer with very few capabilities, including its access to DIS, and while VISIT has some interface capabilities, it is limited by its Oakridge protocol. Merging these two interfaces into a single unit provides both labs with an interface that is DIS compatible as well as increasing the capabilities for the user. Furthermore, any advancements in the interface from one lab could be easily implemented in the other and vice versa. The overall effect of having an integrated graphical interface is that both labs will have expanded capabilities, including the ability to interact with each other in a combat simulation scenario.

This research focuses on finding a method for converting BattleSim and VISIT to DIS, then implementing that method. Furthermore, it integrates VISIT and BattleBridge into a single entity.

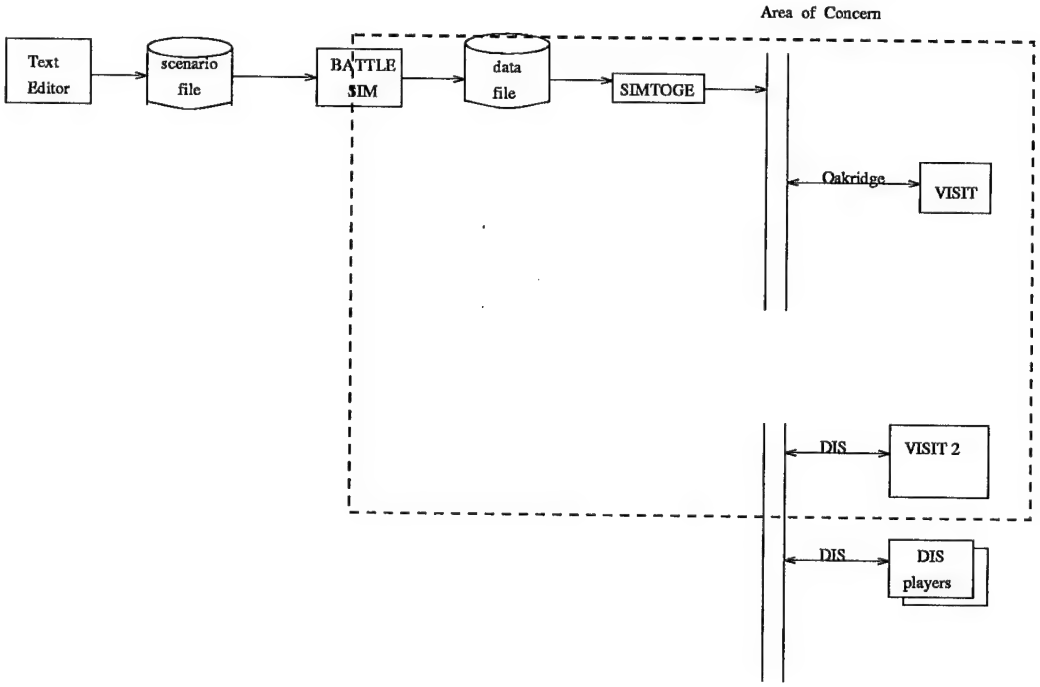


Figure 1. AFIT Parallel Simulation

To help the reader gain a greater feel of what has been discussed so far, a "big picture" was created. Figure 1 shows this overall "big picture" with the dashed-line showing the area of concern. The area of concern contains those systems and applications that were examined and/or modified during this effort. It depicts the environment as it existed prior to the start of this research. VISIT communicates with the network via the Oakridge Protocol. The picture also shows BattleSim running in one of its three modes. As a note, the third mode produces no output so the picture appears to have BattleSim running in one of only two modes. The third mode is simply not depicted in this picture. The picture also shows the Synthetic BattleBridge (SBB) communicating via the DIS protocol, along with several other players. The break in the network is to signify that VISIT and BattleSim were unable to communicate with other DIS players.

**Problem Statement:**

*Design an architecture that incorporates VISIT and the existing synthetic BattleBridge graphics system and demonstrate the ability to provide simulation of a large number of forces in the real-time DIS environment. Furthermore, integrate BattleSim into the DIS network so that it is capable of interacting in a real-time DIS environment.*

### *1.3 Project Objectives*

The object of this research is to develop an object-oriented package that can utilize the features of both VISIT and the Synthetic BattleBridge. Furthermore, this package will be executable on any Silicon Graphics (SGI) machine.

There are two goals that support the objective to be met in this research. The first goal is to reuse as much code as possible from both VISIT and the SBB. The second goal is to design the system so that it can be easily ported from one laboratory or computer system to the next.

### *1.4 Initial Assessment of Past Effort*

This research is based on the research of Looney (7) and DeRouchey (3). The current simulator is designed so that BattleSim is the primary processor of the action on the field. A scenario generator composes routes which are fed into BattleSim. BattleSim, using a discrete event simulation design, processes the events until a final result is determined. VISIT is used to graphically view the events as they are executed in BattleSim. Also, BattleSim has been extended so that it can save its state at an interval specified by the user. This allows the user to stop the simulation, back up to a specified time, then let BattleSim proceed with the simulation from that point.

VISIT and BattleSim are excellent platforms for investigating simulations of warfare. The discrete event simulation format makes time dependent upon the event, not vice versa. This means that a simulation dependent on time completes one loop of execution during one specified unit of time. That unit may be one second in length, or some fraction of that second. A simulation that is dependent on event execution completes one loop of execution during each event. These events may occur every second or every five minutes. The key is that the time lapse between events varies whereas the time lapse between time units remains constant. However, event dependent systems, like VISIT and BattleSim lack the ability to handle real-time interaction. Whenever the user wants to stop the system, BattleSim must stop its processing (which is usually well ahead of what the user is watching), and backtrack to the state saved just prior to the point the user requested. As a result, BattleSim has to undergo some modifications before it is ready for real-time interaction.

## *1.5 Scope*

The focus of this research effort is to design a system that utilizes code already generated for BattleBridge and VISIT. By doing so, this system has the ability to be used on any Silicon Graphics (SGI) platform. However, it also examines possible enhancements to VISIT itself. Such enhancements include the ability to port the system to a Sun SparcStation or a Peronal Computer and the ability for the icons to follow terrain. However, it is not concerned with increasing the realism of the simulation graphics.

## *1.6 Approach*

To meet the proposed research objectives, the following approach was followed:

1. *Gain an understanding of graphics simulation* - Review current literature that details the use of graphics in simulation, then evaluate those documents for possible implementation with the VISIT graphics system. Also, gain a greater understanding of the DIS protocol and examine ways of converting BattleSim and VISIT.
2. *Analyze the VISIT graphics system* - Perform an in depth study of VISIT and BattleSim. Compare the results to an analysis of the BattleBridge and identify any parallels that could result in the production of a single graphics system.
3. *Convert BattleSim to the DIS protocol* - Develop a plan for smoothly transitioning BattleSim from Oakridge to DIS, then implement the plan.
4. *Convert VISIT to DIS by integrating it with BattleBridge* - Determine the differences between the two interfaces, develop a plan for integrating them into a single interface, then implement the plan.

5. *Verification of System* - The system will be tested to ensure that it works correctly and that no deficiencies have been introduced.

6. *Show Value of System* - Determine how the new interface and DIS implementation have improved the accuracy and usability of battlefield simulator.

## *1.7 Assumptions*

Prior to conducting this research effort, it is necessary to state a key assumption. It is assumed that every system involved, including VISIT, BattleSim, and the BattleBridge, incorporate good object-oriented design principles. By doing so, it was easier to develop an architecture that can accomodate both systems, along with any future systems.

# *II. Literature Review*

## *2.1 Introduction*

Enhancing the Visual Interactive Simulation Interface Tool, hereafter refered to as VISIT, so that it is compatible with a common network communication protocol builds on the work of others relating to six different systems known as VISIT, BattleBridge, DIS, BattleSim, the World State Manager, and the Common Object Database. Knowledge of the strengths, weaknesses and limitations of each is necessary for producing a valuable addition to the selection of graphical interface tools available on the Distributed Interactive Simulation. This review provides the baseline knowledge of the existing systems and provides a common starting point for the reader before commencing with the primary focus of this thesis investigation. To help round out the foundation, this review also examines the IRIS Performer System.

Sections 2.2 and 2.5 describe two separate graphical interface tools that have been developed prior to this effort, VISIT and the Synthetic BattleBridge. A distributed simulation communication standard, known as the Distributed Interactive Simulation (DIS), upon which BattleBridge is based, is described in Section 2.3. Section 2.4 briefly explores the World State Manager (WSM) and the Common Object Database (CODB), both of which are used by applications in the Graphics Laboratory to interface with the network. Section 2.6 describes a battlefield simulator, called BattleSim, which provides the information for VISIT to interpret and represent. Finally, Section 2.7 examines IRIS Performer to gain a greater understanding of the system on which the Synthetic BattleBridge (SBB) is based.

## *2.2 Analysis of VISIT*

In 1990, DeRouchey designed and implemented the Visual Interactive Simulation Interface Tool, VISIT (3). VISIT was designed to provide a visual representation of the output generated by the battlefield simulator, BattleSim. Part of the user support provided by VISIT includes the ability for the user to stop, start, initialize, and restart the simulation. VISIT runs on a Silicon Graphics workstation using a local communications package. This package, known as the Oakridge Protocol, was developed by Oakridge National Labs and in conjunction with AFIT. It is used almost exclusively within the Parallel Processing Laboratory. As a result, the usefulness of VISIT is limited due to its inability to interact with any battlefield simulator not using the Oakridge Protocol.

VISIT was written in C and DeRouchey's design involved using modules where each module has a specified purpose. These include interfacing with the network, updating the screen output, and receiving input from the user, along with several other miscellaneous chores. There are 5 modules in VISIT, along with 36 header files. These header files contain information that is common to several modules. Instead of retyping the information, it is consolidated into a single header file and those modules requiring the information access it by including that file in their list of include files.

When DeRouchey first started his design, he followed many principles of object-oriented design. He organized the key functions into a small number of modules and was pursuing a similar approach as he incorporated the header files. However, as his work progressed, more time was spent in building VISIT than refining the design. As a result, the level of maintainability and portability decreased. This is evident by the lack of structure in the header files. As noted in the introduction,

there are 36 header files for only 5 modules. There is no rule of thumb for number of header files per module, but a 7:1 ratio does imply a weak design which often results in a poor structure.

When DeRouchey reviewed his requirements, he determined that one of the non-functional requirements of his research was that the graphical interface would run on a Silicon Graphics Iris 4D workstation. Later, he determines a Silicon Graphics Iris (SGI) 4D/85GT will be used because it "utilizes the UNIX operating system and provides a software development environment using the C programming language" (3). Furthermore, this system also "utilizes special graphics hardware and a complete graphics library" (3). Later, DeRouchey addresses the issue of using other computer platforms for his work. However, according to previous thesis work, none of them matched the SGI 4D/85GT in both performance or capability.

This dependency on SGIs limits the number of users who are able to use VISIT. Since numerous users utilize a Sun SparcStation or a Personal Computer (PC), VISIT is not a viable means for them to interact with a simulation. Looney opted to not address this issue in his work so this SGI dependency is still a limitation to VISIT.

Figure 2 shows how VISIT communicates with BattleSim when the two systems are directly linked. BattleSim, executing on the Hypercube, sends its output to Visithost. Visithost checks the timestamps and, if the user-specified time interval is met or exceeded, saves the state of the simulation. The data is then sent, via ethernet, to VISIT on the SGI. Any user commands are sent, via the ethernet and Visithost, directly to BattleSim. If a rewind command is sent, Visithost restores the BattleSim to the state just prior to the specified time. BattleSim is designed to reset its state without Visithost. Once the simulation is reset, it waits for the user to send the 'play' command before continuing.

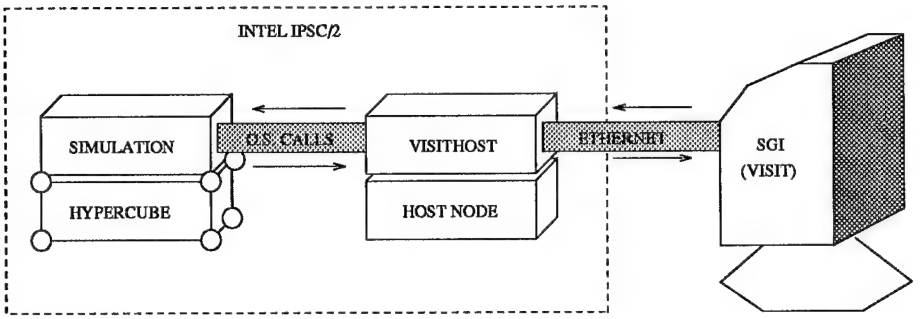


Figure 2. System Overview

VISIT receives asynchronous time-stamped events from the simulation representing movement of players. Since the simulation is event-driven it is necessary to use dead reckoning to provide a constant movement of video. Dead reckoning involves using a player's known speed and heading, then extrapolating its next position based on that information. This approach can result in errors. The system may "dead-reckon" a player to a position different from its actual position. This occurs due to messages arriving later than scheduled. Without the new message, VISIT continues to update a player's position based on its last known speed and direction. As a result, the user may see various objects hopping as VISIT makes corrections to locations. Also, VISIT assumes that events will arrive before their scheduled execution time. Since VISIT only displays what it receives from BattleSim, any missing or delayed messages are a result of BattleSim, not VISIT.

A player is an entity that is participating in the simulation and is represented by an icon. Most of VISIT's icons are jet fighters, tanks, or missiles. Other icons have been created, but are rarely used. At the present time, VISIT is set to use only 200 players (7). This value is only limited by the amount of memory available on the SGI. VISIT stores these players in a storage buffer that is designed, once it reaches the end, to wrap around to the beginning. However, it lacks overwrite protection. Therefore, with simulations using more than 200 players, the risk is very high that the status of some players would be overwritten. The impact of using more than 200 players is that

player 201 and up will overwrite player 1 and up to the number of players over 200. This occurs because VISIT wraps around to the start of the array when it tries to access the next record after the last player. As a result, those players who are overwritten are removed from the simulation erroneously. This action would then cause inaccurate results for the simulation.

Each object's identification index is numbered sequentially with the first index being set to 0 (7). This design has the advantage of direct access to a player, but it does result in increased memory requirements, especially for small simulations. Furthermore, it also lacks flexibility and independence between the application and the interface.

For performance reasons, VISIT stores the entire simulation in a buffer in main memory. It serves to keep track of the data and prevents its loss should BattleSim run faster than VISIT. In the original design, the buffer was used to store the simulation data so that the user could simply replay the scenario from that instead of running BattleSim again. However, as a result of Looney's effort, the buffer is used mostly to ensure data integrity as the simulation is carried out.

The output generated by VISIT is three dimensional color graphics. The icons and the terrain used are simplistic in nature. Each icon is a set of points and colored polygons and the terrain is a series of hexagons quilted together to provide a basic playing surface. As discussed in Section 2.5, BattleBridge employs more intricate and informative icons and terrains that are based upon a system called Performer, discussed in Section 2.7.

In 1993, Looney added the ability to stop a simulation, rewind it to a specified time point, then replay the scenario from that point (7). This is done by having BattleSim save its state at a given time interval, which is specified by the user. Then, to replay a scene in the scenario, the user selects the **stop** command from the menu interface in VISIT. That command is sent back to

BattleSim via Visithost. From there, the user selects the **restart** command and enters a desired simulation start time by using the display interface in VISIT. The time is sent to BattleSim, which resets the states of the players to that point in time, then begins execution from that point. This feature allows the user to review a scenario, or any aspect of it, as often as desired. Since the user lacks the ability to alter the state of an entity in the middle of a scenario, he can review the scene several times and change the original scenario file. Then the user can review the changes and observe the impacts of the differences made in the scenario file.

### *2.3 Distributed Interactive Simulation*

The Distributed Interactive Simulation (DIS) was designed to allow multiple simulation systems to interact with each other (8). In order to do that, it was necessary for DIS, which is an IEEE (Institute of Electrical and Electronics Engineering) standard, to become world recognized (10). Another key to its success is that no central computer is required to arbitrate among the various players. This results in lower overhead, greater flexibility, a more stable system, and even slight improvements in response time. In a simulation environment where there are numerous interactive systems and a single processing computer, the processing computer can become bogged down with the heavy workload. Mostly because it is responsible for tracking the data needed by every system involved with the simulation. By not using this central computer, all overhead requirements are placed on the systems involved instead. The network is just responsible for the transfer of Protocol Data Units from one location to the next, which explains the increase in speed of the simulation.

Communication is handled via Protocol Data Units (PDUs). There are several types of PDUs, with the most common ones being the Entity State PDU, the Fire PDU, and the Detonation PDU. These PDUs contain information such as the Player ID, the Icon ID, the Player Label, and other

information that is applicable to that player. With these PDUs, each player is responsible for maintaining its own state and for determining the impact, if any, of any actions taken by other players.

As described earlier, dead reckoning is also used by DIS. However, due to the newness of dead reckoning to simulation, there is still work being done on determining which algorithm is best.

### *2.4 Analysis of World State Manager and Common Object Database*

The World State Manager (WSM) and the Common Object Database (CODB) are two applications used in the Graphics Laboratory in support of the other DIS applications developed there. The World State Manager is used to access the network and read or write PDUs to/from the network. The data from these PDUs are stored in the CODB. From here, the other applications access the CODB to obtain the data they need.

Prior to execution, each application requiring the use of the WSM and the CODB execute their own version of those two programs. From there, those versions are dedicated to supporting that application as it interacts with other players and systems on the network. The WSM and CODB continue running until the supported application finishes its execution.

### *2.5 Analysis of BATTLEBRIDGE*

The Synthetic Battlebridge (SBB) was first implemented by Vanderburgh in 1994 (11). It is designed to retrieve entity information from other DIS players and present it in a graphical format that engulfs the user into the simulation as the battle progressed. To immerse the user, a pod was designed to assist the user in moving around the battlefield. The pod consists of 6 panels, each of

which has a variety of functions. A function is accomplished by pressing a button on the panel. The functions include general movement of the pod, the ability to instantly move to a given player or place, statistics of the given player, and a simple radar. All user inputs are received via mouse and keyboard with the mouse being the primary input device.

As it stands, there are several drawbacks to the pod. The first problem is that the user's movement is limited by the pod controls. Although it is possible to jump from landmark to landmark, general movement is based upon a series of up, down, left, right, forward, backward, and rotate buttons. The end result is pod movement that is very cumbersome and, since BattleBridge lacks a rewind/playback capability, all but the most adept users risk missing critical actions of the simulation. Another flaw is that the user's view is often hampered by the panels themselves since the user must "see" through them in order to see the battle. Several panels have a hide button which causes the rest of the buttons on the panel to become invisible. This feature helps, but it isn't on every panel and it hampers the user's ability to quickly relocate the pod. The third problem is the pod's inability to rotate around the user. In many cases, the user will orient himself next to an entity and will want to watch its status while watching the entity. There are two methods for doing this. The first is to rotate the user so that he can see the status panel, but he can't see the player. The second approach is to rotate the pod away from the player so that the panel desired is facing the entity. Once this is done, the player can rotate himself back so he can watch both items. Of course, now he must rotate back to the movement panel if he wishes to view another aspect of the object.

The feature that the SBB is noted for is the detail in its icons. These icons are generated by another program known as the Designer Workbench (DWB) which allows the students in the

Graphics Laboratory to reach a level of detail that is unattainable with a standard graphics system. The SBB uses Performer as its base which handles the loading and manipulating of these icons. Unlike the icons in VISIT, which employs basic polygons, the icons in the SBB are painted in camoflage colors and the terrain employs texture to give it a more realistic appearance.

To help enhance the SBB, code was written to utilize a Head Mounted Display (HMD). Wearing the HMD gives the user a feeling of being in the simulation itself. The battlefield is very large and, in some instances, it is very difficult to see every icon on the map. Therefore, large three-dimensional objects were created to help the user see the icons. A large football represents an airplane while giant pyramids represent ground forces. A large stripe, blue for friendly forces, red for enemy forces, on one end of the football indicates heading and side identification of the aircraft. Likewise, one side of the pyramid is color coded to indicate direction and side of the ground vehicle. When the user moves closer to a football or pyramid icon, the football/pyramid is replaced with the icon itself.

The SBB was written in C++ and is based on a player class concept. In this, every player type is defined as a class. Each class has a set of characteristics and a set of functions to manipulate the values of those characteristics. Like VISIT, the BattleBridge runs in a polling loop that looks for updates from the network and the user. As the BattleBridge is a DIS application in the Graphics Laboratory, it is dependent on the WSM and the CODB, as described in Section 2.4, for obtaining information from the network.

There are 46 modules and 70 header files that compose the SBB. As stated earlier, the SBB incorporates player classes which, by nature, have one module and one header file per class. This leaves 24 header files not associated with classes and most of them are standard header files which

are commonly found in C and C++ libraries. The structure of the SBB is much more modularized than VISIT, but due to its large size, it is very complicated to decipher. As a result, maintenance is very difficult and porting the SBB from one system to another is a time consuming process.

There are several positive features of the BattleBridge. As stated already, the most prominent feature is the detail in the icons and the terrain. Another positive feature is that the SBB is reported to be able to handle over 1000 players in a single simulation, although this has not been tested. The final positive aspect is the replacement of the pod system of controlling functions with the less obstructive Head Mounted Display.

### *2.6 BattleSim*

BattleSim is an object-oriented battlefield simulator and uses the Oakridge Protocol to communicate with VISIT. Currently, it is the only simulator at the Air Force Institute of Technology (AFIT) using that protocol. BattleSim is based on a Parallel Discrete Even Simulation (PDES) architecture (6). The time synchronization method used by BattleSim is a conservative method based upon the protocol developed by Chandy and Misra (1, 2). This method means that a processor will execute an event at time $\tau$ only if it has received inputs from every possible source and if every event received is scheduled for execution at time $\tau$ or later.

Scenarios for Battlesim are designed using a scenario generator, or a text editor, where each object is given a specified route to follow. These entities deviate from their routes only when another object is detected. Once this occurs, an entity may react in one of several ways: attack the detected object, evade the object, or ignore the detection entirely and continue on its programmed path.

BattleSim operates in three modes. The first mode results in no output from the simulator and is often used for analysis and maintenance. The second mode lets VISIT use the output directly from BattleSim via the Oakridge Communication Protocol. This mode is the preferred mode of operation. The third mode causes BattleSim to write the output to a file. This file is then read by SIMTOGE, which sends the data to VISIT using the Oakridge Protocol.

There are two methods available for VISIT to receive the data needed to display simulation. The first method is a direct link from the BattleSim to VISIT. The second method is from a data file, usually called 'display.out,' which is sent to a file converter, called SIMTOGE (SIMulation to Graphic Engine). SIMTOGE converts the data into a format recognized by VISIT, then sends that information to VISIT. This mode is often used for debugging VISIT.

When Looney added the ability to rewind and playback a scenario, several modifications were made to BattleSim. This included adding the ability for BattleSim to record, restore, and free an object state. Part of the process of initializing BattleSim involves setting the time between saving states. The user may modify this value during the simulation or opt to use the default value. When the simulation time is 0, the system performs a record object state for each object, then sends an event to the event queue that will tell the system to record these states again at the end of the given time interval. BattleSim continues doing this until the user decides to stop and rewind the simulation. This is done by having the user select the **stop** command followed by the **restart** command. Visithost gets this command from the user and tells BattleSim to restore the object state at the given point in time. As BattleSim restores the state of each object, it frees the object states that were saved after that time, thus freeing up memory space in the system. All states of an

object that are saved are stored in a linked list. This makes it easier to rewind to previous states without requiring a lot of memory in advance.

### *2.7 Analysis of IRIS Performer*

The IRIS Performer package is a product of Silicon Graphics Computer Systems. Its primary purpose is to "extract maximum performance from SGI machines for visual simulation, virtual reality, game development, and high-end CAD systems" (4). This system is designed to work in conjunction with SGI platforms, particularly the Onyx RealityEngines.

Performer works by storing the scene created in a tree format, where a parent is an object or a group of objects. Correspondingly, a child is an object or a component of that object. By doing this, it gives the user the ability to isolate scenes so that effects such as light and shading can be applied across an entire group instead of adjusting each class. Performer is also organized so that it handles all matrix calculations; the user is simply required to supply the pointer to the class or group and the new values for that class/group.

A feature of Performer that adds to its appeal is that it uses instancing in its scenes. According to Hartman and Creek, "Instancing is a powerful mechanism that saves memory and makes modeling easier" (5). Performer utilizes two versions of instancing, shared and cloning. In a shared instance, several parent nodes point to the same child node. Any changes to that child are reflected in each parent. In a cloning instance, a child is copied so that, although several parents may point to similar children, a change in one child does not affect the other children or parents. This feature gives users greater flexibility in designing their simulations.

The key advantage to using Performer is that it is easy to generate simulations that are based upon existing models of vehicles, structures, and terrain. The disadvantage is that it must be used as a skeleton with all non-Performer code built around the Performer commands and functions. This makes it harder to modularize the Performer code which is necessary to do in order to alter the type of graphics package the user wants to use.

## *2.8 Conclusion*

The purpose of this review was to examine those systems and concepts that play a role in advancing the development of the graphical tool, VISIT. The parts that were covered included the graphical tools, VISIT and BattleBridge, and a network communication protocol, DIS. The review also covered details of BattleSim, the Battlefield Simulator for which VISIT provides the graphical interface.

VISIT and BattleBridge both have positive qualities such as the rewind/playback capability of VISIT and the realistic objects and terrain found within BattleBridge, but the most profound difference relates to the ability to widely use each interface.

The Distributed Interactive Simulation (DIS), currently used by BattleBridge, is a communication protocol that is recognized by IEEE and is a standard used around the world. It offers a higher degree of flexibility and versatility than the limited Oakridge Protocol currently used by VISIT. Reconfiguring VISIT to utilize DIS greatly enhances its usability and practicality.

The World State Manager (WSM) and the Common Object Database (CODB) are used to interact with the network, which allows designers to focus more on the intricate features of their

applications and relieves them of the worries with network communication. The ony concern for the engineer is to incorporate the interface needed for accessing the information stored in the CODB.

BattleSim is a battlefield simulator that also uses the Oakridge Communication Protocol. Unlike many simulators, BattleSim uses a Discrete Event Queue system so that it is event driven, not time driven. Like VISIT, BattleSim has to undergo some modifications so that it can interact with other systems using DIS.

Finally, IRIS Performer provides an easy means for generating realistic scenarios. The downside to this system is the dependency that the rest of the code must have on Performer to work.

# *III. Requirements Analysis*

## *3.1 Introduction*

The requirements for this research are generated by the parallel simulation research faculty and staff at the Air Force Institute of Technology. This chapter describes the required functionality for an interactive parallel simulation system research tool.

There are two requirements for this effort with the first being to integrate VISIT with the Synthetic BattleBridge (SBB) based on the Distributed Interactive Simulation (DIS). The second requirement is to integrate BattleSim with DIS. There are three supportive requirements that result from these requirements. The first is to support stop and rewind PDUs. These are used to handle the stop and rewind function in VISIT and BattleSim. The second supporting requirement is to be able to interact with all DIS-based applications in the Graphics Laboratory and the Parallel Laboratory. Accomplishing this requirement ensures that the new graphical system is 100% DIS compatible. The final resulting requirement is to convert SIMTOGE so that it produces its output in DIS PDUs instead of the Oakridge Packets it was producing.

The graphical interface system is a composite of three major modules with two modules decomposing into more submodules. The first module serves as the core graphical engine. It contains those variables and functions common to both VISIT and the SBB. These include tracking the characteristics of each player and making any necessary updates. The next module consists of the tools needed to interact with the network. This module is dependent mostly on the Common Object Database (CODB) and the World State Manager (WSM). These systems are responsible for sending and receiving PDUs to and from the network. This module only reads and updates

those players that the graphical tool is observing. The final module is responsible for the graphics portion of the system. Its two submodules deal with Input and Output.

The Input submodule decomposes into two interface types, menu and pod. Both are mouse driven, but they differ in complexity. The menu is the simplest design and was designed so that it only appears when the user holds down the left mouse button. From there, the user makes his selection. The pod is more complicated and is always seen throughout the simulation. The pod consists of six panels with each panel containing a series of buttons and/or displays, as described in Section 2.5.

The output module consists of two different rendering systems, Standard and Performer. The Standard system uses tools that can be found on nearly every computer platform. Each icon is defined by a file which contains the points needed to generate that icon. The Performer system generates more realistic graphics, but the libraries required for it are not as common as the standard libraries.

### *3.2 System Operation*

In consideration of the requirement for portability, it is necssary for the system to operate in one of four modes. The mode is determined by the graphic methods used along with the type of interface being used.

*3.2.1 Mode 1: Performer with Pod.* In this mode, the graphical tool behaves in a manner similar to the current SBB. Both aspects, the input and the output, relies on the Performer system.

*3.2.2 Mode 2: Performer with Menu.* This mode displays the output via Performer calls. However, the input is received from the user by a standard menu system. As stated earlier, this menu is a floating menu that appears whenever the user clicks the left button of the mouse.

*3.2.3 Mode 3: Standard Graphics with Pod.* Standard graphics refers to every icon used being a file of points, which are drawn in a specific order to create the desired object. These objects lack the detail of those objects found in the Performer system. However, the libraries required to generate them are more widespread, thus increasing the portability of the system. The pod being used in this mode is similar in design to the pod used in Mode 1. The difference is that it is generated in the same manner as the icons, with simplistic graphics.

*3.2.4 Mode 4: Standard Graphics with Menu.* This mode should prove to be the simplest mode available. The graphics, as described in Section 3.2.3, are simple in nature. Likewise, the menu being used in the mode is of a simple design, as described in Section 3.2.2.

*3.2.5 Miscellaneous Modes.* The discussion so far has only covered the use of the graphical tool on Silicon Graphics Machines. There is also a user requirement for running this tool on a Sun SparcStation or a Personal Computer (PC).

## *3.3 Maintainability and Portability*

Two key principles of successful software engineering are maintainability and portability. A system that has been properly designed can be easily maintained and used on several different systems with minimal effort.

According to Schneidewind, maintainability is "the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements. A key to successful maintenance is to design a system so that it can be easily modified." (9) Both VISIT and the SBB were implemented upon weak design plans. As a result, neither one can be properly maintained without some extensive effort from the maintainer. Part of this effort is to design a system that is easier to maintain and modify.

The other issue involved is portability. Both systems, VISIT and the SBB, suffer severe portability problems. One of Looney's last efforts was to try and compile, then run VISIT on a SGI in the graphics lab (7). This effort failed, but the reasons for the failure were never reported. In a similar fashion, the SBB is not portable either. However, unlike VISIT, the reason for this problem is known. The SBB is highly dependent upon Performer calls in order to work. These Performer libraries are rarely on SGIs in AFIT other than those found in the Graphics Laboratory. As a result, the SBB can only be used in the Graphics Lab. This research also develops and implements a means for making this graphical tool portable between various systems.

## *3.4 Review of Possibilities*

Analysis of the system discussed in Chapters 1 and 2 lead to three possible ways of meeting the requirements as they are laid out.

*3.4.1 Option 1.* The first approach is to enhance the BattleBridge by incorporating the features of VISIT. This can be done by simply adding the rewind/playback ability possessed by VISIT. An advantage to this approach is that the DIS protocol is already in use so no conversions are required. However, there are several problems with this approach. First of all, it still lacks

maintainability and portability. The SBB design utilizes very few principles of software engineering which makes it difficult to maintain or modify. Since this approach is dependent upon the SBB for its success, these negative traits would still exist in the new system.

Another problem would be in setting the SBB up so that it can recognize when it is working with BattleSim only. The reason for this is that the rewind/playback feature is only available when the system is interacting with BattleSim alone.

The third problem is SBB's lack of portability. Due to the fact that it is highly dependent on Performer, which is rarely available on other systems, this graphical tool is limited to which systems can run it.

*3.4.2 Option 2.* The second approach is to modify VISIT so that it can handle the graphics of BattleBridge. This approach eliminates the need to add the rewind/playback ability, since it already exists in VISIT. However, it is still necessary to add the DIS protocol to it. Furthermore, like the SBB, VISIT is lacking in several key software engineering principles. Although it isn't as complicated as the SBB, VISIT is still a difficult program to maintain and update. Furthermore, it lacks the ability to port to other systems. However, on the bright side, the graphics used by VISIT can be used on any SGI system.

*3.4.3 Option 3.* The final approach is to design an architecture that is optimal in every area. This architecture incorporates the key features of each system, which include the ability to rewind/playback a simulation and the ability to be DIS compatible. Furthermore, this architecture is portable from SGI to SGI and from SGI to any other graphics system. These include Sun SparcStations and Personal Computers. The advantage to this approach is that it already possesses

the rewind/playback capabilities and no conversion is required for it to use DIS. Furthermore, this code will be more maintainable and portable, especially considering the possibility that unneeded code will be removed. To switch between graphics systems and interface types, it will be necessary to change a line of code for each, then recompile the program.

### *3.5 Option Selection*

A close examination shows that the third option is the most feasible one to pursue. The primary reason is that it keeps the functionality of both systems with minimal, if any, modifications. It also provides the greatest chance of achieving portability and maintainability in the system. An area of concern is that it is highly dependent on the reusability of the old code. Any problems in that code will be transferred into the new system. Another concern is that this approach may add more modules to the overall system which could impact the portability of the system. However, with proper design techniques, it is possible to smoothly incorporate the new code as well as eliminate any code that is no longer used.

# *IV. Design*

## *4.1 Introduction*

This chapter describes the design used in creating the architecture as defined in Chapter 3. Appendix B contains a detailed description of VISIT and the Synthetic BattleBridge (SBB). It is broken down by component with a brief description of each.

Based on the information contained in Appendix B, Section 4.2 describes the design of the new architecture. It also points out where in the design the key components of VISIT and the SBB are incorporated. The chapter ends with Section 4.3 which points out and briefly describes the concerns with implementing this architecture.

## *4.2 The New Architecture*

With both VISIT and the SBB broken down into their primary modules, the next step is to design the architecture that incorporates all these pieces into a single system, which is referred to hereafter as VISIT 2. With this thought in mind, the architecture takes the form of a skeleton with the various modules described in Sections B.2 and B.3 serving as the body. In an effort to keep the design simple, it is necessary to group the overall functions of the system into three primary areas of responsibility: a network interface, a central storage/processing unit, and a basic input/output system. Figure 3 shows the new architecture design used for the new system.

*4.2.1 VISIT 2 Header File.* This file serves as a central point for the three groups to reference for type definitions, global constants, and in-line functions, along with other key elements that are common throughout the system. As part of this module, a new player class is created that consists of two variable types, which are kept hidden from the user, and numerous functions,

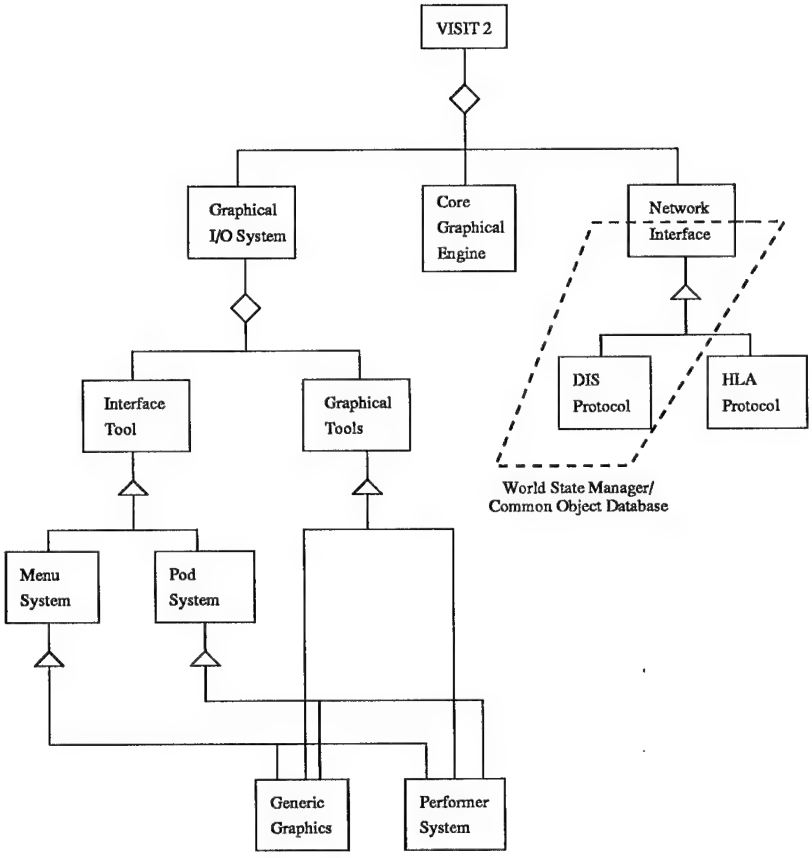


**Figure 3. New Architecture Design**

which are for public use, that are used to modify the hidden values. The two variables are defined by the object types used in VISIT and the SBB. By doing this, it is not necessary to "integrate" these types into one single type. The primary advantage is that it is not necessary to rewrite the graphical functions which already work with the two different graphics packages. Determining which value type of the player depends on which graphics system is used, standard or performer. In addition, this approach makes it easier to include object types from any other system that are to be incorporated in the future.

To support this new player class and the rest of the modules in VISIT 2, the following files are included in this header file:

1. *VISIT.h* - This header file contains type definitions that are standard throughout the VISIT program, along with the object type used in the player class.
2. *AFITcom.h* - Along with the communication types, this file also contains other values required throughout the system.
3. *Geomfile.h* - This file contains constants that are needed for handling file input/output along with conversions of PDUs into standard records for internal use.
4. *Player.h* - This is the type definition for a player in the SBB which is incorporated into the new player class defined earlier.
5. *Clock.h and YA_Clock.h* - Both of these files are the type definition for time, both real and simulated, for the entire system.
6. *Sim_Models.h* - This file defines constants that are used throughout the SBB and VISIT 2.

7. *Sim_Entity_Mgr.h* - The primary purpose of this file in the SBB is for indexing the players in the system. VISIT 2 tracks its players in a slightly different manner. However, this file is included because it contains parts of the type definition in the new player class.
8. *Event.h* - As described in Section B.3.4.1, this file defines the events encountered by the players. It is included here since it is used by both the Network Interface and the Core Graphical Engine.
9. *VISutils.c* - This module contains generic tools that can be used anywhere in the VISIT 2 system.

The body of this header file handles the initialization of VISIT 2, the loop which keeps the simulation running, and program termination. As a result, the two modules, *VISmain.c* and *main.cc*, are not used directly, but serve as references for this system.

*4.2.2 Network Interface.* The network interface is responsible for reading PDUs from the network and converting them into a format that VISIT 2 can recognize. This module also takes information from VISIT 2 and converts it into PDUs which are written to the network. Since the World State Manager (WSM) and the Common Object Database (CODB), as described in Section 2.4, are so successful in interfacing with the network, this design depends on them for actual network interaction. The figure also shows the use of a Higher Level Architecture (HLA) protocol. However, since HLA is not finalized for public use yet, it will not be discussed any further. The point in including it in Figure 3 is to show that should it, or any other network protocol, be approved for use with Air Force applications that this architecture is designed to incorporate that new protocol.

From here, the next step is determining which modules and header files are grouped into this module. Review of Sections B.2 and B.3 readily show that the following files belong in this area:

1. *Base_Net_Player.h* - This file defines several PDU types that are used and is also useful in converting the PDUs to a file format recognized by VISIT 2.

2. *DIS_v2_Entity_Obj_Mgr.h* - In the SBB, this file handles the interaction between entities and PDUs. In VISIT 2, this file assists in the conversion of PDUs to the format recognized by the Core Graphical Engine.

As explained earlier, VISIT 2 relies on the WSM/CODB system for network interaction. Therefore, no VISIT files are used since they utilize the Oakridge Communication Protocol. Those SBB files that are included serve as a means for converting PDU data into a temporary player record which simplifies the modification of players in the Core Graphical Engine.

*4.2.3 Computer Graphics Engine.* The Computer Graphics Engine (CGE) serves as the workhorse for the entire system. It is responsible for storing and tracking the players, including the creation, modification, and destruction of each player. It utilizes the functions defined in the player class to accomplish this task. In addition, the following files are used:

1. *Math.h* - This file contains fifteen calls to external functions which are useful in manipulating the data in the VISIT player type.

2. *Objects.h* - There are four external function calls which are used to modify how the user sees a player, including scaling, translating, and rotating that player.

3. *VISobjects.c* - This module contains functions that map players to their icon files, construct the players prior to displaying them on the screen, and updates the values of the players. In addition, it contains the bodies of the prototypes described in *Objects.h*.

*4.2.4 Graphics System.* Of the three modules, this is the most complicated. As is seen in Figure 3, this module is broken down into two submodules, both of which are broken down further into two more submodules. The first submodule, Interface Tools, is described in Section 4.2.4.1 while the second submodule, Graphical Tools, is described in Section 4.2.4.2. The only file this submodule needs to reference is the header file described in Section 4.2.1.

*4.2.4.1 Interface Tool.* This submodule is primarily responsible for handling input from the user. Since VISIT and the SBB both sport different interfaces, this module is designed to switch from one interface type to the other, based on the capabilities of the platform and the desires of the user. The user can switch between the two types by setting a defined value switch in the header file described in Section 4.2.1. The only files that are incorporated into this module are *mouse.h* and *mouse.cc* since the mouse is the most common input device for either system.

*Menu System.* This module allows the user to interact with the simulation without using the complicated pod. By holding down the left button, the user makes the menu appear so that he can make a selection from it. The only file that is used here is *VISmessage.c* because it is already designed to handle the menu system.

*Pod System.* As can be seen in Section B.3, there are numerous modules that make up the pod in the SBB. Each name in the following list represents both the header file and the module that use that name. Furthermore, unlike the previous sections, no further explanations will be given in this section as to why a particular file is in this list since it is already known that each one represents a necessary piece of the pod. The only exception to this is *LabFont.h* which was included because it controls the font size of the labels used in the pod.

| | |
|---|---|
| *Bottom_Panel_Type* | *Bottom_Panel_Controls_Type* |
| *Button_Type* | *Center_Panel_Type* |
| *Clouds_Pod_Interface_Type* | *Common_Pod_Colors.h* |
| *Drop_Camera_Hud_Interface_Type* | *Fod_Pod_Interface_Type* |
| *Grid_Hud_Type* | *Grid_Pod_Interface_Type* |
| *Hide_Sub_Panel* | *Hud_Interface_Type* |
| *LabFont.h* | *Left_Panel_Type* |
| Left_Panel_Controls_Type | *Locators_Trails_Threats_Type* |
| *Moonroof_Pod_Interface_Type* | *Panel_Type* |
| *Panic_Type* | *Pause_Type* |
| *Pod_Hud_Type* | *Pod_Player_Type* |
| *Right_Panel_Type* | Right_Panel_Controls_Type |
| *Site_To_Site_Pod_Interface_Type* | *Weather_Type* |
| *Sub_Panel_Type* | *Top_Panel_Type* |
| Top_Panel_Controls_Type | *Video_Missile_Hud_Interface_Type* |
| *Zoom_Type* | *Translation_Type* |
| *Rotation_Type* | *SBB_Simulation* |
| *Attached_to_Player_Info_Type* | *Attach_to_Player_Pod_Interface_Type* |

The reader needs to be reminded that the pod is written so that it utilizes Performer only. A recommended modification, that is described in greater detail in Section 6.3.1.1, is to build a pod based solely on a standard graphics package. This allows the user to select an interface type based on desire, not on limitations of the graphics package.

*4.2.4.2 Graphical Tools.* This component is responsible for displaying the simulation output to the screen. Since the user may opt to use a standard graphics package or the performer system, this module was designed to easily switch between the two modes. This is done by setting a flag in the header file described in Section 4.2.1, then recompiling the system. The following files were incorporated in here since they can be utilized with either system. Since they are dependent on Performer, some modifications are required so they can operate with the standard graphics system.

1. *Net_Stats_Type.h*

2. *Site_to_Site_Functionality_Type.h*

3. *Weather_Functionality_Type.h*

4. *Drop_Camera_Player.h*

5. *Video_Missile_Player.h*

*Generic Graphics.* This module utilizes the generic graphics package found on most SGI platforms. Since VISIT is based on this package, the following files are from VISIT only.

1. *Geomdefs.h* - There are three external function calls that are needed for displaying icons with the standard graphics package.

2. *Graphics.h* - This file contains five external function calls useful for altering the user's view on the simulation.

3. *Misc.h* - This file contains eight external function calls that are used to start and stop drawing figures with the standard graphics package.

4. *VISgeomfile.c* - This module reads in the data files for the generic icons.

5. *VISdisplay.c* - This module contains the functions needed to utilize the tools in the standard graphics package.

*Performer System.* This module is responsible for handling the graphics by using the Performer package. This system is not ready for use now since a means for modularizing the Performer functions calls completely has not been discovered. By modularizing this system, it

makes it possible to swap this package out for the user to employ the standard graphics package or any other package that has been modularized for use in this architecture.

Figure 4 graphically shows how the modules detailed in Appendix B are incorporated into the new architecture.

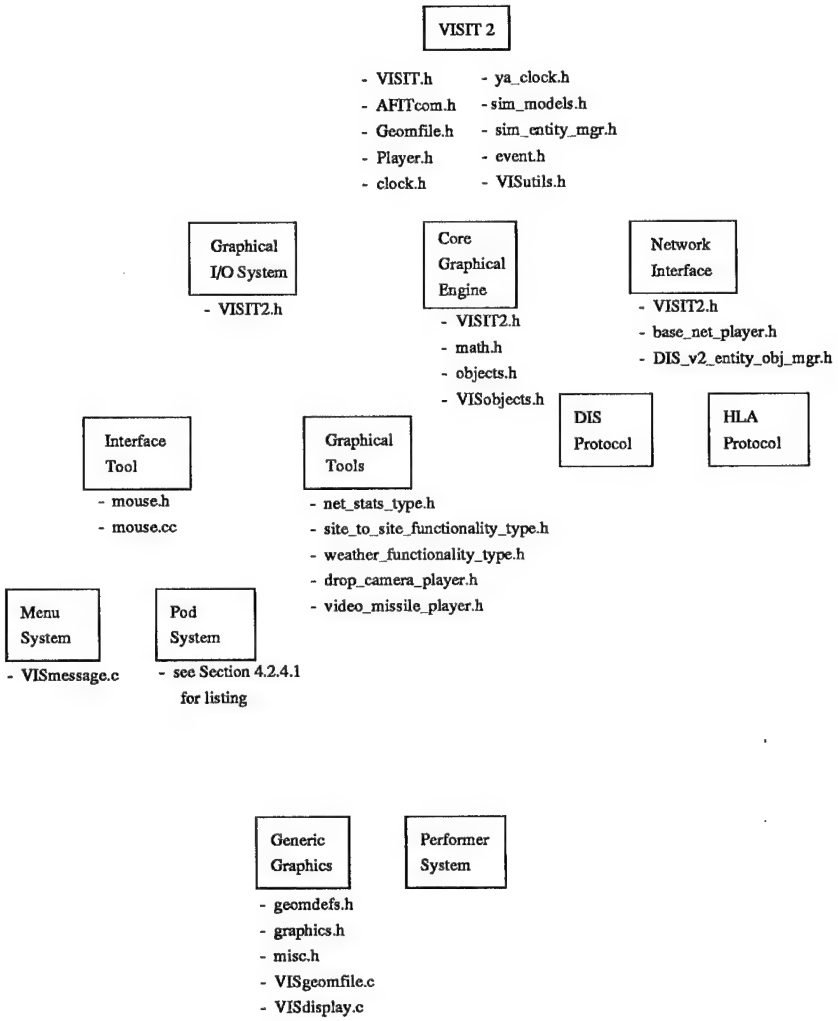


Figure 4. Dependencies in VISIT

### *4.3 Concerns in the Design*

There are two concerns with this design that have not been addressed yet. The first relates to the large number of external calls made by modules and header files in VISIT. The bodies to most of these function calls exist within the modules defined earlier in this chapter. However, there are several whose function bodies have not been located yet. The makefile, which utilizes a C compiler, has no trouble finding these bodies, but a C++ compiler, with which this architecture is compiled, is more particular about knowing the exact location of these bodies prior to linking the object code.

The second concern is that of integrating two different graphics packages that are utilized in two different programming languages. Although C++ is an advanced version of C, there are still some differences that impact the ability to reuse code. Furthermore, as stated in Section 4.2.4.2, a means for modularizing the performer code into a single module has not yet been discovered. This modularization is a key aspect of this architecture and could impact the success of this design.

# *V. Test Results*

## *5.1 Introduction*

This chapter details the test plan used to determine the success of the architecture. Due to linking errors encountered during the coding of the program, none of these tests were carried out.

## *5.2 Test Plan*

Each test case is broken down into three areas: name of the test case, the purpose of the test case, and how the test case is planned to be carried out.

*5.2.1 Testing of Design During Coding.* This process was an on-going process as the system was coded. As each major component was developed and integrated, the system was tested to ensure that the new component worked and that the functionality of the old components still existed. Most of this testing was accomplished by comparing text output with expected results. The following paragraphs detail the test plan scheduled to be used during coding of the modules in the order that the modules were to be integrated.

*5.2.1.1 File Loading Test using Standard Graphics.* This test was used to ensure that the system was loading the icon files used by the standard graphics package correctly. This was done by calling the required function, then printing out the results. From here, a visual check was done to ensure the success of the module.

*5.2.1.2 Graphical Still Shot Test using Standard Graphics.* The purpose of this test was to ensure that the generic graphics package still worked as designed. The test is very simple and

the expected result is a scene with the terrain and a single icon being displayed. After a successful completion of this test, then the test will be repeated with multiple icons displayed.

*5.2.1.3 Graphical Motion Test using Standard Graphics.* This test takes the previous test a step further by loading the single icon and having it move across the terrain. The data is generated from hand and hardcoded into a test module. Once this test is completed, then it will be repeated with multiple icons. This will ensure that the system is ready to handle the flow of information from the network.

*5.2.1.4 Loading Performer Player Test.* This test involves the performer graphics package. The object is to create a dummy player, then have it appear on the screen by using the performer package. In addition, the system would display this character's attributes by printing them out onto the screen. A visual comparison of the display window and the text output would confirm the success of this function.

*5.2.1.5 Graphical Motion Test using Performer Graphics.* This test is similar to the test case described in Section 5.2.1.3, except that it involves the Performer graphics system. All data is hardcoded into the program, then executed, with the results displayed on the window and all attribute characteristics printed to the screen. A visual check of these two windows will confirm the success of this test.

*5.2.1.6 Network Input.* This test was accomplished in two parts. The first step involved disabling the graphical portion of the system and having it print the results to a file instead. The data used is generated from a PDU generator known as Gaggle. This program is run from a SGI in the Graphics Laboratory and dumps large amounts of PDUs to the network for

150 entities. These PDUs are nothing more than Entity-State PDUs, which only provide position updates. A visual comparison of the output from Gaggle and this system will determine the success of the test.

Once the first step of this test is accomplished, the next step is to enable the graphical portion of the test. With the graphics in place, this test will be repeated as above, except the results will be compared with the results of the SBB. This will help to ensure that both systems are "seeing" the same thing.

*5.2.1.7 Menu Interface Test.* This test runs in a fashion similar to the previous test. The first step will allow the user to enter commands from the mouse controlled menu. However, instead of having an impact on the screen, the results will be sent to a data file. This file will be visually checked to ensure that the commands were received correctly and that the proper action was carried out. The next step will be to set the system so that the user inputs will be enacted on the screen. To ensure functionality of the entire menu system, only one command will be tested at a time and it must pass before the next command will be implemented. This requirement will apply to both steps of this test.

*5.2.1.8 Rewind/Playback Test.* One of VISIT's primary abilities was to stop a simulation at any point, rewind to specified point, then restart the scenario. This test was broken down into two parts. The first consisted of having the user send the command via the menu interface to the Network Interface module. This module would use a PDU to send the request to Visithost. From here, Visithost would merely display a message saying that it received the message.

The second part simply consisted of commenting out the message displayed by Visithost and allowing it to stop, rewind, then restart the simulation as designed. The steps described in the previous paragraph would be used to test this step.

*5.2.2 Final Testing.* This test plan involved the testing of every functionality in the system. This included the ability to obtain data from the network to displaying it on the screen to receiving input from the user to sending information back to the network.

This testing process assumes that the test plan as detailed in Section 5.2.1 is successful. Furthermore, the primary purpose of this test is to ensure that every functionality of the system works properly. If any functionality fails, then this test is expected to find it so that it may be corrected.

# *VI. Conclusions*

This chapter reviews the overall research effort by examining those requirements and goals that were satisfied. Section 6.2 examines those requirements and goals that were not met along with possible reasons why these were not achieved. The chapter then concludes with Section 6.3 which discusses future recommendations and final thoughts on the project.

## *6.1 Satisfied Requirements and Goals*

As described in Chapter 3, one of the requirements was to design and implement an architecture that integrates VISIT with the SBB and is based on DIS. This requirement was met in that an architecture was designed that integrated both graphical systems so that each one utilized the DIS Communication Protocol.

## *6.2 Unsatisfied Requirements and Goals*

The second requirement was to integrate BattleSim with DIS. This requirement was not met due to difficulties encountered with implementing the design for the first requirement. The plan to accomplish this requirement was based on the success of the implementation of the architecture. By doing this, it would be possible to create an interface that converted BattleSim records into PDUs which would be written to the network. In addition, this interface would read PDUs from the network and convert them into a record format recognized by BattleSim.

Another part of this effort that was not accomplished was to convert SIMTOGE so that it produced PDUs instead of Oakridge Communication Packets. The success of the task depended on

the successful completion of integrating VISIT 2 with the WSM/CODB. As was described earlier, this effort was not completed successfully.

## *6.3 Conclusions*

Although a final product was not created, this effort was still successful. As requested, an architecture was designed to accommodate the sponsor's requirements while leaving it flexible enough to support future modifications.

### *6.3.1 Recommendations.*

*6.3.1.1 Future Modifications for the Pod.* The pod was written based solely on the Performer Graphics System. There is a lot of functionality in the pod that a user will find helpful during a simulation. A recommended special study or aspect of another thesis effort would be to design a similar pod based solely on the Standard C/C++ Graphics package. By doing so, this would give the user the ability to interface with the system either by the pod or the menu system, even if the user remains limited by the standard graphics system.

## *Appendix A. Definitions and Acronyms*

| | |
|---|---|
| **AFIT:** | Air Force Institute of Technology |
| **ARPA:** | Advanced Research Projects Agency |
| **BattleSim:** | Battlefield Simulator that uses a PDES architecture. |
| **CAD:** | Computer Aided Design |
| **CODB:** | Common Object DataBase |
| **DIS:** | Distributed Interactive Simulation |
| **DMSO:** | Defense Modeling & Simulation Office |
| **DWB:** | Designer WorkBench; Used to generate icons and terrain used by IRIS Performer. |
| **FAA:** | Federal Aviation Administration |
| **HMD:** | Head Mounted Display |
| **IEEE:** | The Institute of Electrical and Electronics Engineering |
| **IRIS Performer:** | A graphics package written for SGI computer platforms and designed to utilize the hardware in the most efficient manner for best graphical performance. |
| **JWFC:** | Joint Warfighting Center |
| **Oakridge:** | A communication protocol used by two or more systems to interact on a network. |
| **PC:** | Personal Computer |
| **PDES:** | Parallel Discrete Event Simulation |
| **PDU:** | Protocol Data Unit; a package type used in DIS. |
| **Player:** | An entity that is participating in the simulation and is represented by an icon. |
| **SBB:** | Synthetic BattleBridge |
| **SGI:** | Silicon Graphics Iris |
| **SIMTOGE:** | SIMulation TO Graphics Engine |
| **STRICOM:** | Simulation Training and Instrumentation Command |
| **VISIT:** | Visual Interactive Simulation Interface Tool |
| **WSM:** | World State Manager |

# *Appendix B. Review of Components of VISIT and the SBB*

## *B.1 Introduction*

This appendix describes the design used in creating the architecture as defined in Chapter 3. Section B.2 will start by examining the module structure and code dependencies in VISIT, then Section B.3 will repeat this process with the Synthetic BattleBridge. Both systems are very complicated and this step proves useful in decomposing them so that a new architecture can be built from the pieces.

## *B.2 The VISIT System*

As stated in Chapter 2, DeRouchey originally employed good object-oriented design practices. However, as time progressed, DeRouchey slowly became less disciplined in following these principles. Figure 5 shows the modules and header files used in VISIT. The lines represent the dependencies that a module has on another, with the modules near the bottom of the figure being dependent on those modules near the top. As a note, this representation applies to all figures in this chapter. These dependencies include function calls, type definitions, and references to global variables. The following subsections detail the modules used and briefly describe their dependencies.

*B.2.1 VISmain.c.* This module is the primary module in VISIT. It is responsible for initializing, or calling those functions that initialize, VISIT. This module is dependent on seven header files or external calls. In this case, there are two external calls made to functions that are found in another module in Figure 5. One of the functions referenced handles the loop that VISIT stays in while the simulation is carried out.

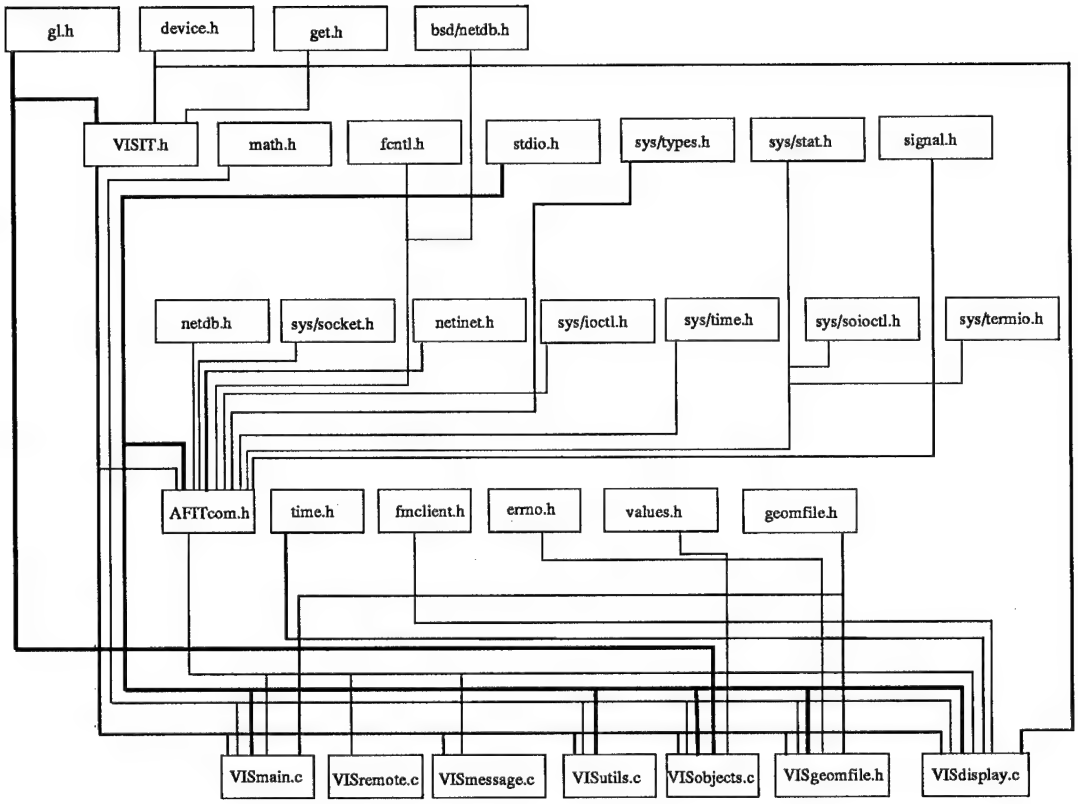


Figure 5. Dependencies in VISIT

Every module, regardless of whether it is a module or a header file, that shows a dependency on *AFITcom.h* and *VISIT.h* are using those files for type definitions. *VISIT.h* contains the definitions common for defining a player while *AFITcom.h* contains types and definitions associated with communicating via the network. In this case, *VISmain.c* has dependencies on both.

Of the remaining three dependencies, two (*math.h* and *stdio.h*) are common to any C library. The first handles common math functions while the second is responsible for input and output, whether it is to/from a file or a screen and keyboard. The final dependency, called *geomfile.h*, contains a few defined constants and in-line macros that are not found in either *VISIT.h* or *AFIT-com.h*.

*B.2.2 VISremote.c.* This module is responsible for opening, communicating via, and closing the socket used to connect with the network. It is dependent on six function calls that are used to open the socket, read from the network, and write to the network. Its only other dependency is on the header file *AFITcom.h*, which is described in Section B.2.1.

*B.2.3 VISmessage.c.* This module has only one function, which is to convert the Oakridge Protocol Packets into a format that the rest of VISIT understands. It is dependent on four calls to external definitions, which are not defined in any module depicted in Fig 4.0. These definitions are located in another file in the system, which is accessed during compiling of VISIT. Like the other modules, *message.c* is also dependent on the two files, *VISIT.h* and *AFITcom.h*.

*B.2.4 VISobjects.h.* This module is responsible for all maintenance of the objects used in the simulation. This includes building the icon, creating the player in the buffer, then displaying the icon on the screen. It also handles updates to the player, including scaling, rotating, and

translating the icon. For debugging purposes, this module also handles printing the data of each player to the screen.

To accomplish these tasks, this module has twelve dependencies. Four of these are based on standard C header files, *stdio.h*, *math.h*, *values.h*, and *gl.h*. These allow it to print data to the screen, handle the mathematical calculations, utilize various floating point types and display the objects on the screen. Furthermore, *objects.h* is also dependent on *VISIT.h* for the type definitions.

The remaining seven definitions are external calls. All seven are variables that are defined elsewhere in the system and has left the C compiler responsible for linking them together.

*B.2.5 VISdisplay.h.* This module is the workhorse of the entire system. Its main purpose is to coordinate all the actions between the modules. In addition, it also handles input from the user. In Section B.2.1, a reference was made to an external function call that starts the polling loop. That function call is found here.

Being the largest module in VISIT, it also has the greatest number of dependencies, twenty-four. Of these, fifteen are external references to variables defined elsewhere in the system. Like the other modules, responsibility for finding these variables is left to the C compiler.

Six of the remaining dependencies are on standard C header files. These include *stdio.h*, *math.h*, *device.h*, *sys/types.h*, *fmclient.h*, and *string.h*. The first two give the module the ability to handle all input and output requirements along with any mathematical calculations necessary. The third file is used to obtain input from the mouse while the fourth file defines types useful for getting maximum efficiency from SGI hardware. The header file *fmclient.h* defines various fonts for text printed in a graphics window and the last file allows the module to manipulate string types.

## *B.3 The SBB System*

In comparison to VISIT, the Synthetic BattleBridge (SBB) is much larger and more complicated. This is evidenced by the fact that it required six figures to map out the 120 modules and header files. To maintain reader interest, modules that are similar in nature will be described in the same section instead of stepping through each module one by one as was done with VISIT.

*B.3.1 Figure Six.* The following module descriptions pertain to those modules found in Figure 6.

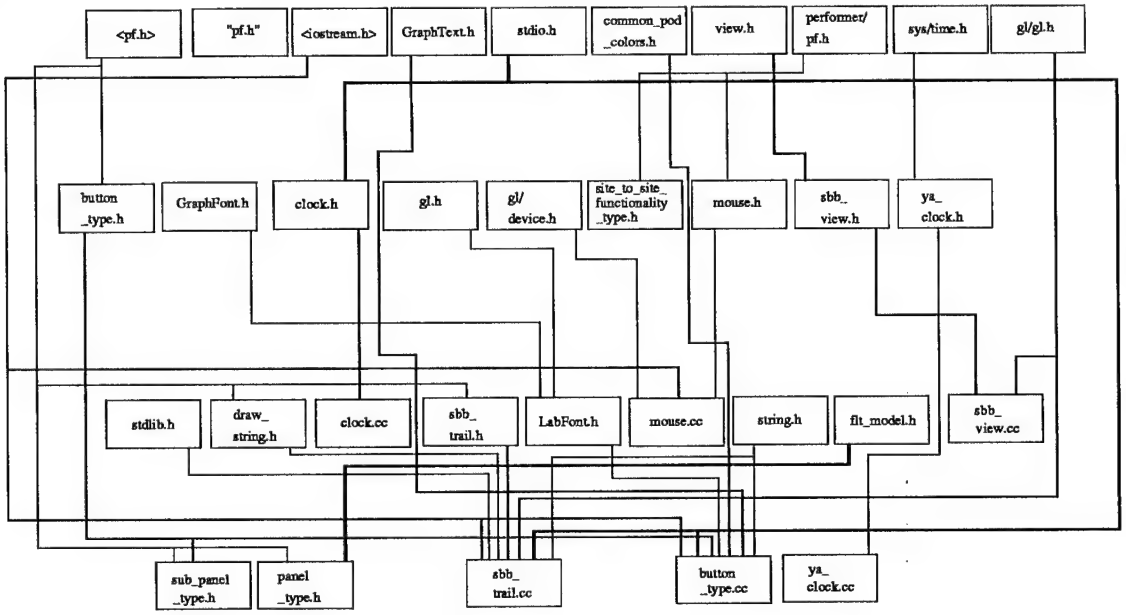


Figure 6. Dependencies in the SBB, part 1

*B.3.1.1 YA_Clock.cc and Clock.cc.* These two modules are similar in that they both track time, but they differ in how they do it and their purpose for tracking it. The module *ya_clock.cc* tracks real-time by obtaining the time from the system clock whereas the module *clock.cc* handles simulation time (i.e., the amount of time that has passed while the simulation was running). Both are solely dependent on their header files, *ya_clock.h* and *clock.h*, respectively.

*B.3.1.2 SBB_View.cc.* This module has the simple task of creating the window on the screen through which the SBB is displayed. It is dependent only on its own header file and the *gl/gl.h* library. The latter contains the commands needed to open the window and name it "Synthetic BattleBridge".

*B.3.1.3 SBB_Trail.cc.* This module is also fairly simple in that it only has the trivial task of drawing trails after objects. This makes it easier for the user to locate icons and to determine their direction at a glance. It is dependent mostly on standard C and C++ header files, which include *gl/gl.h*, *stdlib.h*, *iostream.h*, *stdio.h*, and *string.h*. These files make it possible for *sbb_trail.cc* to handle input/output, to draw the trails on the screen, and to manipulate string types as necessary. The other two files it depends on are *drawstring.h*, which draws strings on the screen at designated coordinates, and *sbb_trail.h*, which serves as the class definition and prototype for the corresponding module class.

*B.3.1.4 Button_Type.cc.* This module plays a key role in the SBB. The SBB is manipulated by the controls in the pod where each control is a button. By pressing a button, the user causes a certain activity to occur, whether it is moving the pod itself or altering the scene as he sees it. This module defines the button, draws it on the screen, and determines its state (on/off).

As a result of its importance, this module is dependent on seven different header files. Among these are *iostream.h*, *stdio.h*, and *string.h*, which are all used to manipulate and display the labels for each button, which are cast as string types. The files *GraphText.h* and *labfont.h* control the fontsize, type and overall appearance of the labels. The remaining files *common_pod_colors.h* and *button_type.h* are responsible for the color of the buttons and serve as the prototype reference to this class, respectively.

*B.3.2 Figure Seven.* The following module descriptions pertain to those modules found in Figure 7.

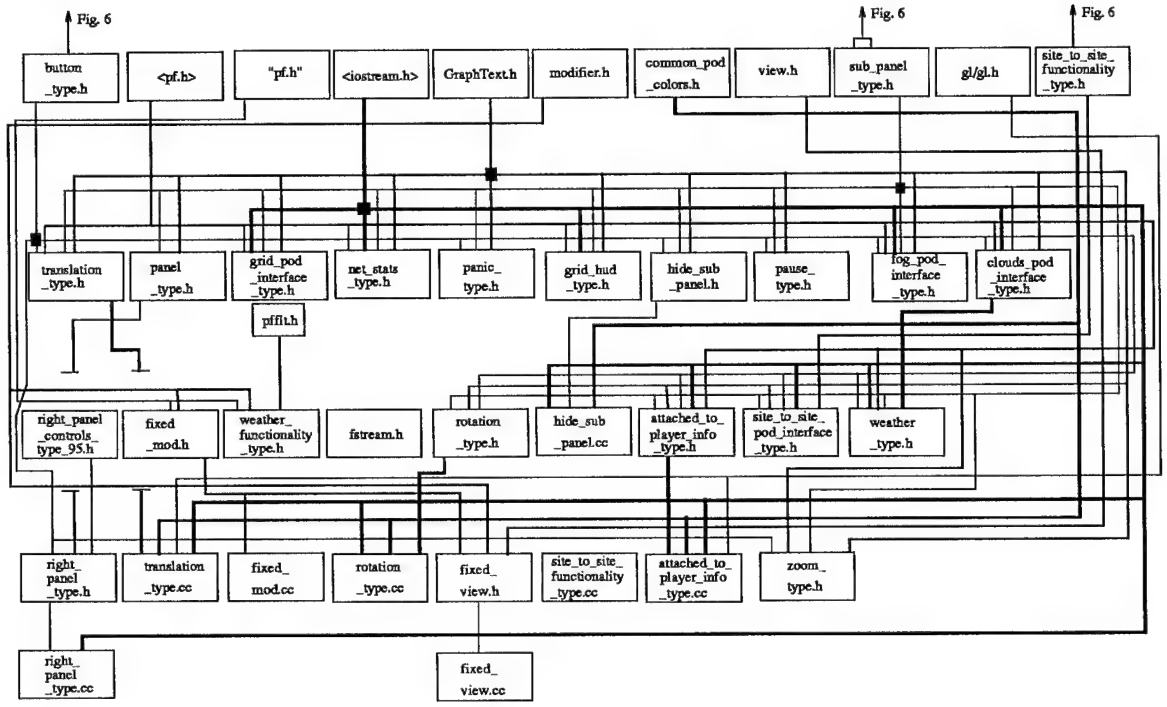


Figure 7. Dependencies in the SBB, part 2

*B.3.2.1 Rotation_Type.cc and Translation_Type.cc.* These two modules are both responsible for movement of the user within the pod. *rotation_type.c* controls the rotational movement of the pod in any direction while *translation_type.cc* controls the movement of the pod from one point to another in two-dimensional space (i.e., left, right, up, and down). Both are dependent on *iostream.h* for handling input and output and on *common_pod_colors.h* which defines the colors of the pod. A key difference in dependencies is that *translation_type.cc* is dependent on *gl/gl.h* while *rotation_type.cc* is not. This header file contains several graphics-related commands that, if are necessary in one, should be necessary in the other. However, no reason has been found for this

discrepancy. The final dependency that each has is on their own header files *rotation_type.h* and *translation_type.h*, respectively.

*B.3.2.2 Hide_Sub_Panel.cc.* Most panels in the pod have a button that gives the user the ability to hide a panel so that he can see the scene clearly. This module is responsible for hiding a desired panel. It tracks the status of the button and the status of the panel, which is dependent on the button. When the button is pushed, the panel is hidden, until the button is pushed again.

This module is dependent on three header files. These include *hide_sub_panel.h*, *iostream.h*, and *common_pod_colors.h*. The first serves as the prototype reference for this module while the second file is used to depict the color of the button. The last header file handles the input received from the user.

*B.3.2.3 Right_Panel_Type.cc.* This module handles user interactions with the right control panel of the pod. It is responsible for tracking which buttons are pushed and for displaying its current status of the panel. To assist with its input and output requirements, *right_panel_type.cc* uses commands found in the header file *iostream.h*. It also has dependencies with its own header file, *right_panel_type.h*.

*B.3.2.4 Attached_to_Player_Info_Type.cc.* This module plays a minor role in the SBB. It allows the user to attach the pod to any player in the simulation. This means that the pod is positioned at a predefined distance, which can be changed by the user, from the player. This allows the user to follow the player throughout the simulation and observe his actions. This module is dependent on four header files, *iostream.h*, *gl/gl.h*, *common_pod_colors.h*,

and *attached_to_player_info_type.h.* The first two provide the functions necessary for handling input/output of data and for displaying any updates of the player on the screen. The third header file is used to control the color of the buttons and text used while the fourth serves as the prototype of this module for other modules to reference.

*B.3.2.5 Fixed_Mod.cc and Fixed_View.cc.* These modules are grouped together primarily because there true function is not known. Both are very small modules and each is dependent only on its prototype header file, *fixed_mod.h* and *fixed_view.h*, respectively. The lack of documentation with these two modules hinders the author's ability to determine their role in the SBB.

*B.3.2.6 Site_to_Site_Functionality_Type.cc.* This module works in conjunction with the *site_to_site_pod_interface_type.cc.* Between the two modules, the user is able to jump from one predefined point to another predefined point within the simulation. This module is dependent on only three header files, two of which are used for input/output from the keyboard to the screen and to/from files. The third header file is the prototype file used to reference this class.

*B.3.3 Figure Eight.* The following module descriptions pertain to those modules found in Figure 8. In some instances, a header file was predefined on another figure. This is noted by having each dependency represented by a line coming out of the top of that file. A single horizontal line is used to connect those lines with a single vertical arrow leading out from it. The figure that defines the header file is noted next to the arrow.

*B.3.3.1 Panel_Type.cc.* This module describes the standard panel baseline. Due to the wide variety of panel formats found in the pod, this baseline is very simple. It defines the

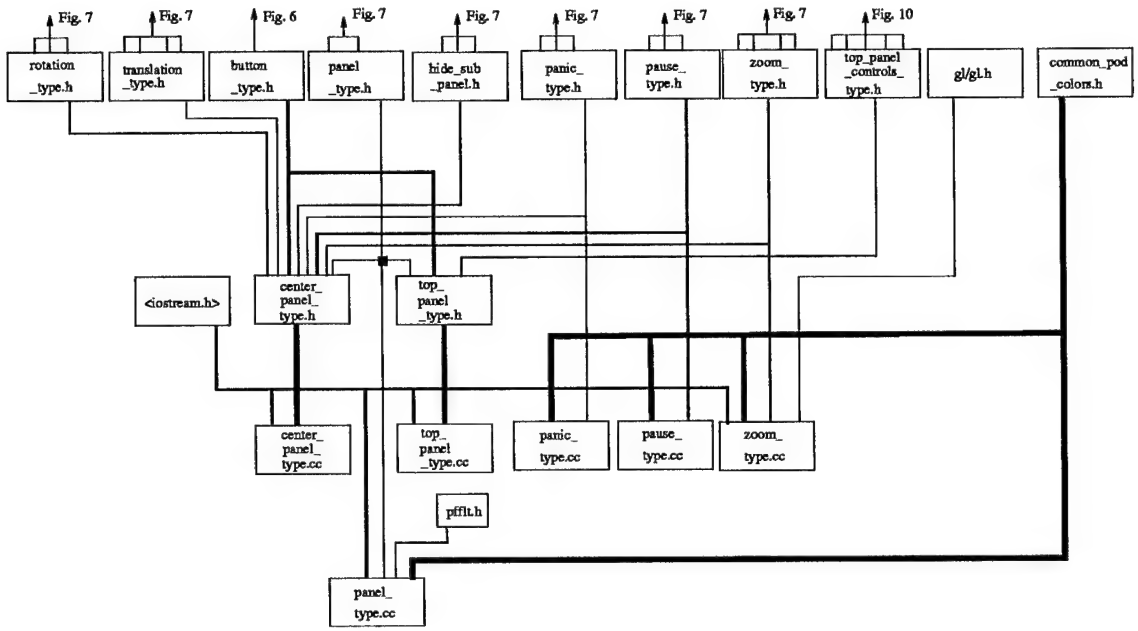


Figure 8. Dependencies in the SBB, part 3

standard outline color and the size of the panel. It is dependent on four header files, *panel_type.h*, *pfflt.h*, *iostream.h*, and *common_pod_colors.h*. The first header file serves as the prototype which is used by the other panel types while the exact nature of the second file is unknown, it is involved with some of the performer functions used in this module. The remaining two files assist the module in handling input/output and with defining the colors used in the pod.

*B.3.3.2 Center_Panel_Type.cc.* This module coordinates the function of the center panel of the pod, which is the most used panel in the entire simulation. This panel handles all movement of the pod, along with pausing the simulation and returning a "lost" user back to a predefined panic point. It is dependent on only two header files. These files are the prototype file, called *center_panel_type.h*, and the input/output file, known as *iostream.h*.

*B.3.3.3 Zoom_Type.cc.* This module allows the user to move forward and backward while inside the pod. The pod can be moved at several speeds so this module also tracks which buttons are pressed so that the pod moves at the correct speed. Two of the header files on which it is dependent are standard C++ header files, *iostream.h* and *gl/gl.h*. As stated earlier, these files allow the module to handle input/output and to utilize graphics functions. The other two header files were generated in the Graphics Laboratory, *zoom_type.c* and *common_pod_colors.h*. The first serves as the prototype which other modules reference while the second defines the colors used in the pod.

*B.3.3.4 Panic_Type.cc.* Through past experience, it was discovered that the user can become disoriented during a simulation. To help a user recover, this module was created. When the button is pressed, the pod is oriented to a heading and pitch of zero, at a predefined point above the horizon. It is only dependent on two header files, *panic_type.h* and *common_pod_colors.h*. The first is the prototype that is referenced by the panel module which employs this button while the second file defines the colors of the pod.

*B.3.3.5 Pause_Type.cc.* This module, as the name implies, allows the user to pause the simulation at any point. It is responsible for maintaining the status of the pause button while setting the flag which freezes the simulation. Like *panic_type.cc*, it only relies on two header files, *pause_type.h* and *common_pod_colors.h*. Both files serve similar purposes as defined by the previous section.

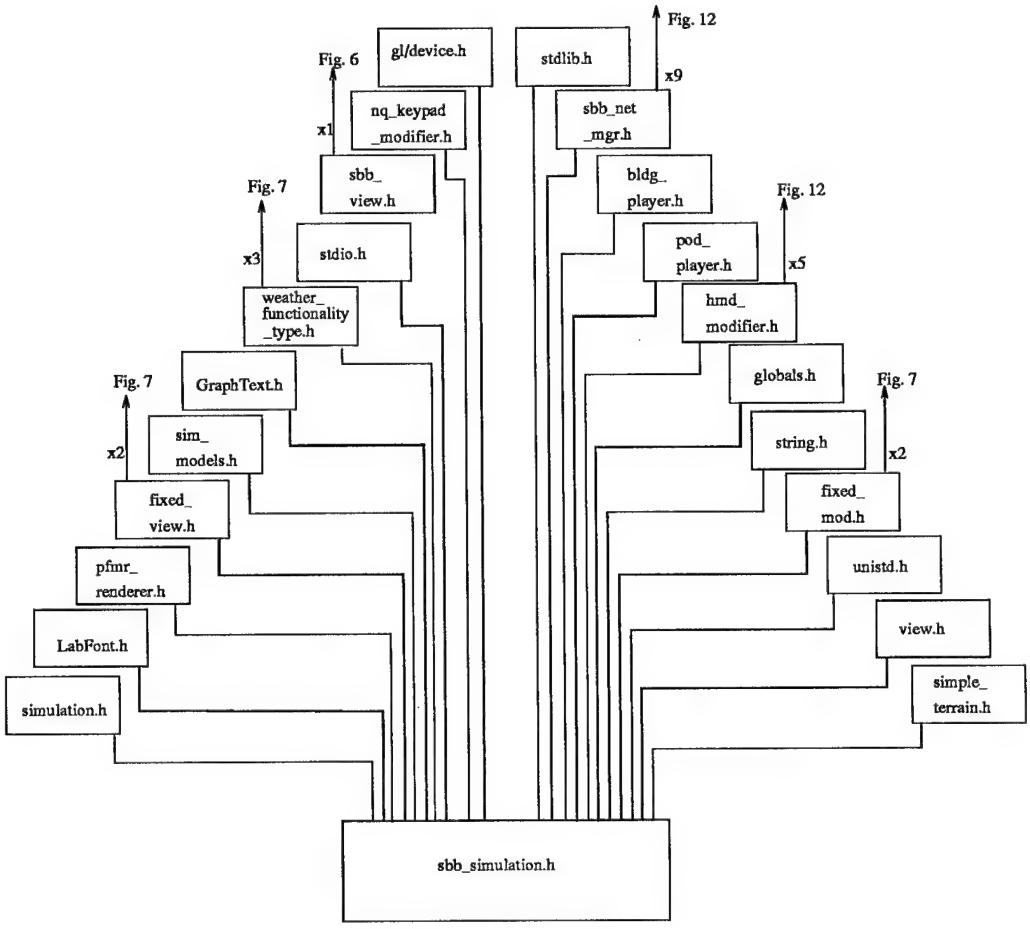


Figure 9. Dependencies in the SBB, part 4

*B.3.4 Figures Nine and Ten.* Figure 9 shows the dependencies of a key header file, *sbb_simulation.h*. This picture is not discussed in this section, but was included for completeness only.

Most of the dependencies shown in Figure 10 are of one header file being dependent on several others. These will not be discussed here, but in sections where a module references that header file.

*B.3.4.1 Sim_Entity_Mgr.cc.* This module manages the entities used in the simulation. This involves everything from their creation to their termination. The module is dependent

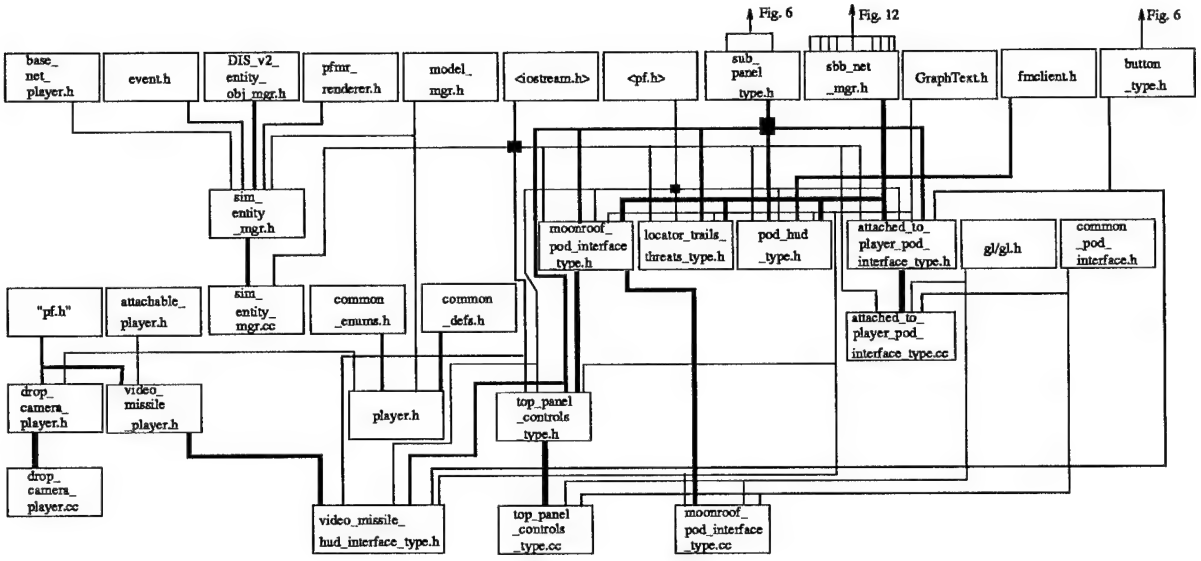


Figure 10. Dependencies in the SBB, part 5

only upon the *iostream.h* and the *sim_entity_mgr.h*, however, the prototype file has several dependencies that need to be noted here. These dependencies include *base_net_player.h*, *event.h*, *model_mgr.h*, *DIS_v2_entity_obj_mgr.h*, and *pfmr_renderer.h*. The first file defines the net player and the PDUS that are used while the second file defines the event types that a player may encounter. The *model_mgr.h* maps an icon, and its path, to the player using that icon and the *DIS_v2_entity_obj_mgr.h* handles the interaction between PDUs and the player entities. The last file is required because every simulation that uses performer requires a renderer. This file binds a performer subclass to the renderer to the simulation using it.

*B.3.4.2 Drop_Camera_Player.cc.* This module defines the class for the remote camera. These cameras can be dropped from the pod so that the user can go back and view the action without having to return there with the pod. This module is dependent only on its prototype file header, *drop_camera_player.h*.

*B.3.4.3 Top_Panels_Controls_Type.cc.* This module handles the position and functions of the buttons used on the top panel. This panel contains the radar, which is described briefly in the following subsection. There are four files that this module is dependent on. These include *iostream.h*, *gl/gl.h*, *top_panel_controls_type.h*.

*B.3.4.4 Moonroof_Pod_Interface_Type.cc.* This function refers to the panel containing the radar set for the simulation. From here, the user can watch the red forces move against the blue forces. It is dependent on four files, *iostream.h*, *gl/gl.h*, *moonroof_pod_interface_type.h*, and *common_pod_colors.h*.

*B.3.5 Figure Eleven.* There are only two modules that will be discussed in this section. Figure 11 shows the dependencies the two files have on other header files. The remaining files that are shown in this figure are header files which will be discussed at a later time.

*B.3.5.1 Bottom_Panel_Type.cc.* This module defines the bottom panel in the pod. The bottom panel is an options panel that allows the user to decide which function it will serve. These options include viewing a radar panel, a weather panel, or a Heads Up Display (HUD). It is dependent on only two headers files, *bottom_panel_type.h* and *iostream.h*.

*B.3.5.2 Pod_Player.cc.* This module defines the pod itself. It is designed as a parent node where its children consist of the panel types. It is responsible for coordinating the every action of the pod and handling every input given by the user. It is dependent on only two header files *pod_player.h* and *pfmr_renderer.h*.

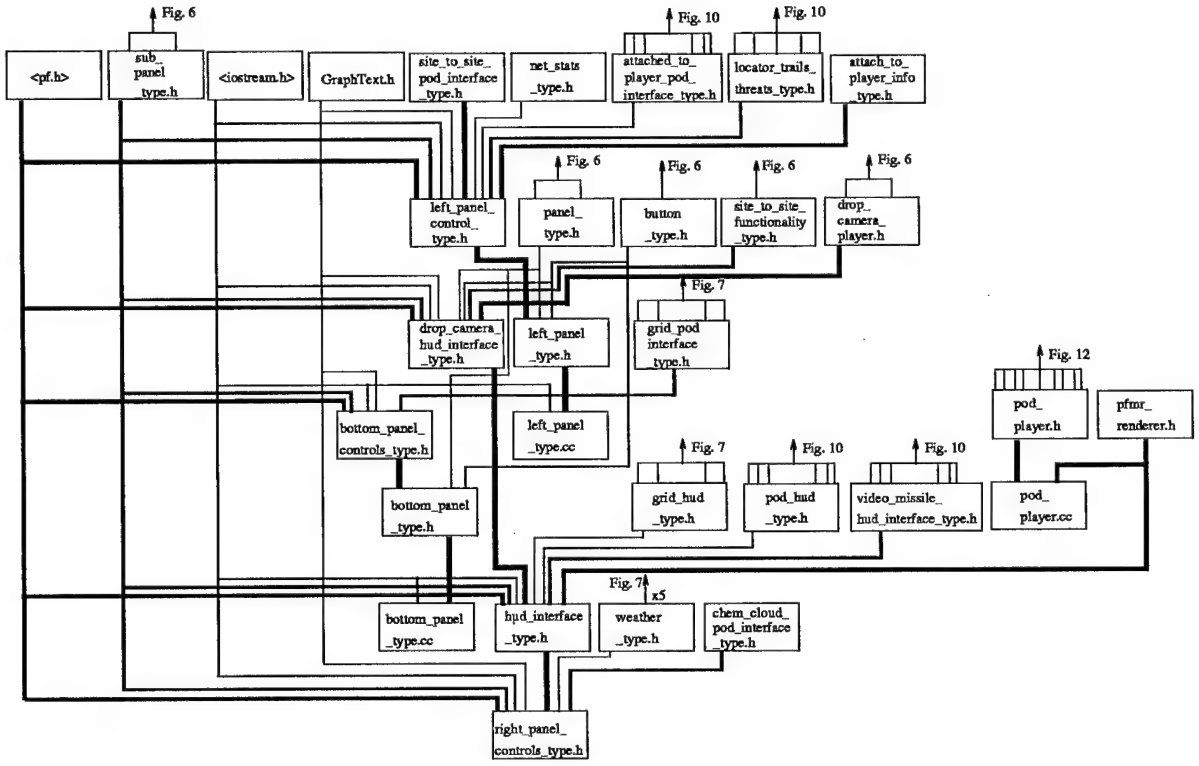


Figure 11. Dependencies in the SBB, part 6

*B.3.6 Figure Twelve.* This figure shows only the dependencies of various header files. These files are not discussed now, but in other parts of this appendix.

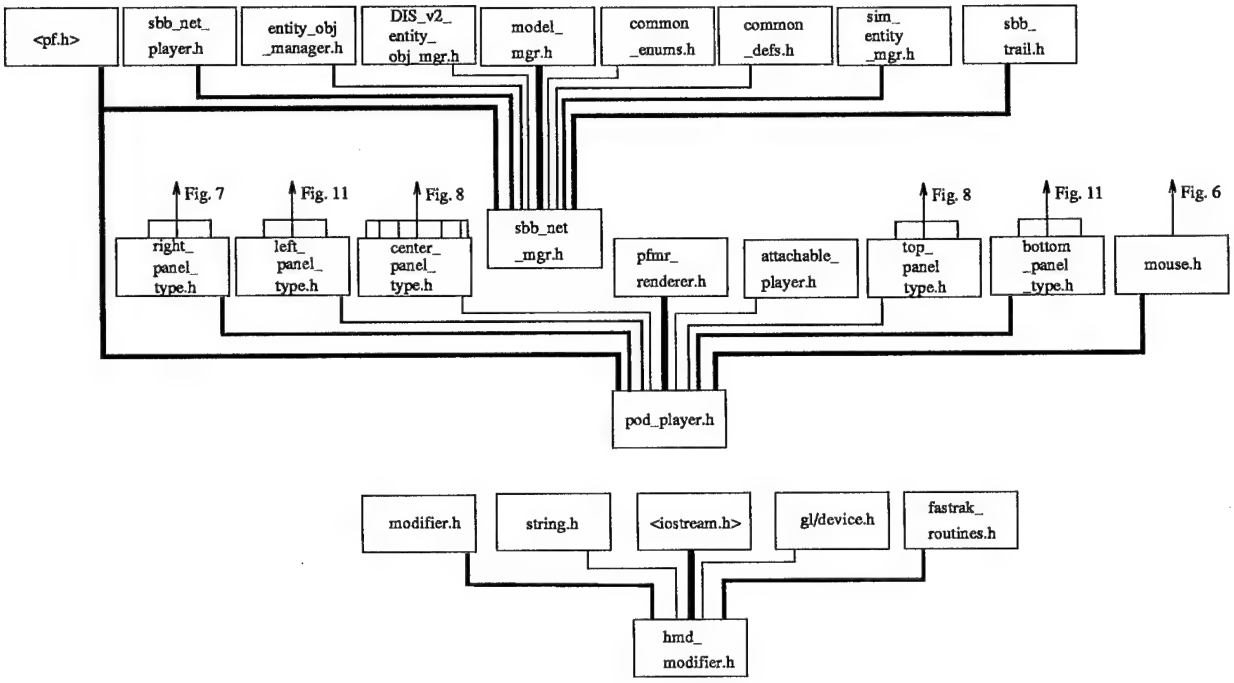


Figure 12. Dependencies in the SBB, part 7

*B.3.7 Figure Thirteen.* Figure 13 contains the final modules with dependencies in the SBB. There are eleven modules that are discussed in this subsection.

*B.3.7.1 Left_Panel_Controls_Type.cc.* This module places buttons along the bottom of the left panel so that the user can select what he wants to see on that panel. The options are a Attach_To_Player_Panel, which allows the user to follow a player in the simulation, a Site_to_Site_Panel, which allows the user to jump from site to site in the simulation, or an options panel, which is described in Section B.3.7.2. It has dependencies on four header files which include *iostream.h*, *gl/gl.h*, *left_panel_controls_type.h*, and *common_pod_colors.h*.

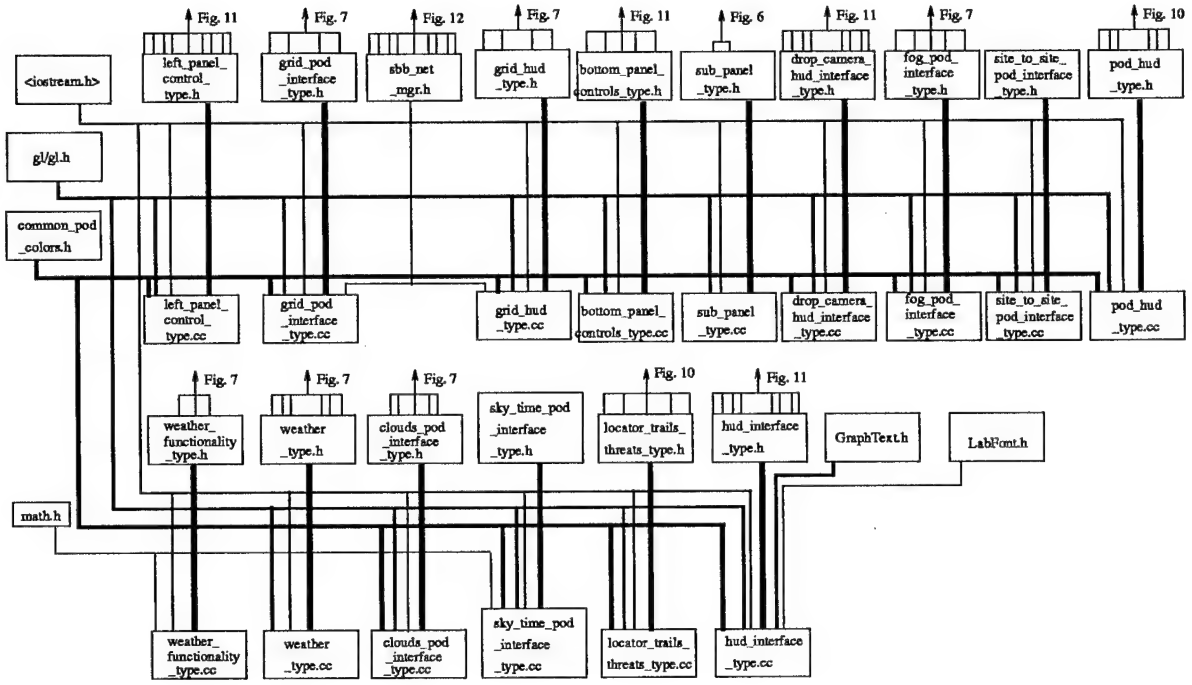


Figure 13. Dependencies in the SBB, part 8

*B.3.7.2 Locators_Trails_Threats_Type.cc.* This module is responsible for drawing four buttons on a single panel. These buttons are labeled as Locators, Trails, Threats, and Ranges. The locators option turns the football/pyramid icons on and off so that the user can find players quickly or not have them interfere with his view. The trails option is designed so that a line follows the players showing where they have been when this option is activated. The threats option shows the threats to other players in the simulation. This is done by having a color-coded cone appear. The color of the cone denotes the side the threat is on. It is dependent on four file headers, *iostream.h*, *gl/gl.h*, *Locators_Trails_Threats_Type.h*, and *Common_Pod_Colors.h*.

*B.3.7.3 Grid_Hud_Type.cc and Grid_Pod_Interface_Type.cc.* These modules are responsible for handling the radar in the top panel, along with the buttons used to control it. The radar uses color coding, red and blue, to denote sides and it has the option to display airborne forces

only, ground forces only, or both forces at once. Both are dependent on five header files, which are *iostream.h*, *gl/gl.h*, *grid_pod_interface_type.h*, *common_pod_colors.h*, and *SBB_net_mgr.h*.

*B.3.7.4 Bottom_Panel_Controls_Type.cc.* This module allows the user to decide what functions appear on the bottom panel of the pod. The options available are a radar panel, a weather panel, or a Heads Up Display (HUD) panel. This module has dependencies on four file headers. These files are *iostream.h*, *gl/gl.h*, *Bottom_Panel_Controls_Type.h*, and *Common_Pod_Colors.h*.

*B.3.7.5 Sub_Panel_Type.cc.* This module is used to define the top of a button. It is designed to determine if the mouse is within the confines of a button when the left mouse button is pressed. It is dependent on three header files, *iostream.h*, *sub_panel_type.h*, and *gl/gl.h*.

*B.3.7.6 Fog_Pod_Interface_Type.cc.* This module, upon a command received from the user, adds fog to the simulation. It is dependent upon four header files which are *iostream.h*, *gl/gl.h*, *fog_pod_interface_type.h*, and *common_pod_colors.h*.

*B.3.7.7 Weather_Functionality_Type.cc.* This module controls the weather in the simulation. It also shows the effects of the weather on the simulation. It is dependent upon three header files, *weather_functionality_type.h*, *iostream.h*, and *math.h*.

*B.3.7.8 Weather_Type.cc.* This module is responsible for the interface that allows the user to alter the weather. This module is also dependent upon three different header files, which include *iostream.h*, *gl/gl.h*, and *weather_type.h*.

*B.3.7.9 Clouds_Pod_Interface_Type.cc.* This module is designed to add clouds to the simulator, along with altering their density. It depends upon four files, which are *iostream.h*, *gl/gl.h*, *clouds_pod_interface_type.h*, and *common_pod_colors.h*.

*B.3.7.10 Hud_Interface_Type.cc.* This allows the user to activate one of three Heads Up Display (HUD) options, or any combination thereof. These options include turning the radar on, activating a remote camera, or activating the video camera on a missile. The radar, which is located on the top panel, is activated when this option button is pressed. When the remote camera button is pressed, the user is given the option to select a remote camera. Once selected, the user views what that camera sees. The third option activates a camera located on a missile, which allows the user to follow a missile as it works its way toward its target. This module is dependent on six header files. These files are *iostream.h*, *gl/gl.h*, *hud_interface_type.h*, *GraphText.h*, *labfont.h*, and *common_pod_colors.h*.

*Bibliography*


1. Chandy, K.M. and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–452 (September 1979).
2. Chandy, K.M. and Jayadev Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (May 1981).
3. DeRouchey, Captain William J. *A Remote Visual Interface Tool for Simulation Control and Display*. MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1990. DTIC: AD-A231 019.
4. Hannan, Shawn Capt. *OBJECTSIM 3.0: A Software Architecture for the Development of Portable Visual Simulation Applications*. MS thesis, Air Force Institute of Technology, 1995. DTIC: AD-A306 137.
5. Hartmann, Jed and Patricia Creek. *IRIS Performer Programming Guide*. Silicon Graphics, Inc., April 1994.
6. Hiller, Captain James B. *Analyitc Performance Models of Parallel Battlefield Simulation Using Conservative Processor Synchronization*. MS thesis, Air Force Institute of Technology, December 1994. DTIC: AD-A289 249.
7. Looney, Captain Douglas C. *Interactive Control of a Parallel Simulation from a Remote Graphics Workstation*. MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993. DTIC: AD-A274 217.
8. Loper, Margaret and Steve Seidensticker. *The DIS Vision: A Map to the Future of Distributed Simulation*. Technical Report, Institute for Simulation and Training, 1994.
9. Schneidewind. "The State of Software Maintenance," *IEEE Transactions on Software Engineering*, *SE-13*(3):303–310 (March 1987).
10. Sheasby, Steven. *Management of Simnet and DIS entities in Synthetic Environments*. MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), 1992. DTIC: AD-A258 921.
11. Vanderburgh, John. *Space Modeler: an Expanded, Distributed, Virtual Envrionment for Space Visualization*. MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1994. DTIC: AD-A289 409.

*Vita*

Captain Glenn G. Jacquot was born February 21, 1969, in Heidelberg, Germany, and was raised in Mesa, Arizona. In December 1991, he graduated with a Bachelor of Science degree in Computer Science from Baylor University and was commissioned a second lieutenant in the United States Air Force.

After graduating from Basic Computer-Communications Officer Training at Keesler Air Force Base in February 1993, he was assigned to the 50th Space Systems Squadron. Here he was responsible for tracking the software systems used for numerous satellite programs, including MilSTAR, DMSP, DSP, and GPS.

In July 1994, he was assigned to the 1st Space Operations Squadron where he handled configuration management responsibilities for the GPS satellite system until he received orders to attend AFIT in May 1995.

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1996 | Masters Thesis |

**4. TITLE AND SUBTITLE**
A Common Architecture for Simulation Viewing over Multiple Protocol Environments

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Glenn G. Jacquot, Captain, USAF

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology, WPAFB OH 45433-6583

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT/GCE/ENG/96D-11

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Capt Mike Lightner
2241 Avionics Circle, Suite 32
WL/AASE BLD 620
Wright-Patterson AFB, OH 45433-7334
(513)255-4429

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*
This study discusses an architecture used for displaying data results in a graphical format. This data is obtained via internet connection or from a direct connection to a battlefield simulator known as BattleSim.
This architecture utilizes several graphics packages for displaying the output along with two different approaches to receiving commands from the user. Furthermore, the design was built to easily incorporate other graphics packages and communication protocols in order to increase its portability.

**14. SUBJECT TERMS**
Distributed-Interactive-Simulation, Simulation, Performer, Graphics, Battlefield, Scenario

**15. NUMBER OF PAGES**
80

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

**Block 1.** Agency Use Only *(Leave blank)*.

**Block 2.** Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

**Block 3.** Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

**Block 4.** Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

**Block 5.** Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | | | |
|---|---|---|---|
| **C** | - Contract | **PR** | - Project |
| **G** | - Grant | **TA** | - Task |
| **PE** | - Program Element | **WU** | - Work Unit Accession No. |

**Block 6.** Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

**Block 7.** Performing Organization Name(s) and Address(es). Self-explanatory.

**Block 8.** Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

**Block 9.** Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

**Block 10.** Sponsoring/Monitoring Agency Report Number. *(If known)*

**Block 11.** Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

**Block 12a.** Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

**Block 12b.** Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

**Block 13.** Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

**Block 14.** Subject Terms. Keywords or phrases identifying major subjects in the report.

**Block 15.** Number of Pages. Enter the total number of pages.

**Block 16.** Price Code. Enter appropriate price code *(NTIS only)*.

**Blocks 17. - 19.** Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

**Block 20.** Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.